PRACTICAL SYSTEMS & CYBERSECURITY EDITION

01001100
01001001
01001110
01010101
01011000

```
student@linux: ~

student@linux:~$ whoami
root

student@linux:~$ cat roadmap.txt
Beginner → Basic → Intermediate → Advanced → Expert
student@linux:~$
```

A COMPLETE HANDS-ON TEXTBOOK & REFERENCE MANUAL

LINUX MASTERY

Beginner to Expert

Ubuntu · Linux Mint · Kali Linux · Ethical Hacking Essentials

Installation · Terminal · Administration · Networking · Security · Bash · Troubleshooting

AUTHOR

Sai

✓ root ✓ sudo ✓ bash ✓ ssh ✓ nmap

How This Book Is Structured

This handbook progresses **Beginner → Basic → Intermediate → Advanced → Professional → Expert**. Read it in order the first time through; use it as a reference afterward. Six recurring box types mark the material throughout:

OB **OBJECTIVE** — What a lab or section will teach you to do	**CM** **COMMAND** — Syntax reference block
TP **TIP** — Efficiency or best-practice note	**WN** **WARNING** — Risk of data loss or system damage
SC **SECURITY** — Security implication or hardening note	**LB** **LAB** — Hands-on practice exercise

ETHICS NOTICE

All security-testing material in this book is written for legal, authorized use only — your own systems, isolated lab VMs, intentionally vulnerable training targets, CTFs, and environments where you have explicit written permission. Nothing here should be used against systems you do not own or are not authorized to test.

Contents

35 chapters · 6 parts

1 Understanding Linux 10
2 Linux Distributions 16
3 Installation 20
4 First Day with Linux 25
5 The Linux Filesystem 34
6 File Management 37
7 Reading and Editing Files 40
8 Users, Groups and Permissions 43
9 Package Management 48
10 Processes and System Management 51
11 Storage 54
12 Networking 57
13 SSH — Secure Shell 61
14 The Bash Shell 64
15 Bash Scripting 68
16 Logs and Troubleshooting 76
17 Linux Security 80

PART I CYBERSECURITY & ETHICAL HACKING 84

18 Linux for Cybersecurity 85
19 Kali Linux for Authorized Security Testing 86
20 Essential Ethical-Hacking Linux Commands 88
21 Wireshark and tcpdump 92
22 Web Security Fundamentals 94
23 Linux Privilege and Security Concepts (Defensive) 96

PART II ADVANCED & EXPERT LINUX 99

24 Advanced Linux 100
25 Expert Linux 103

PART III INTERNALS, LABS & REFERENCE 106

26 Linux Internals: What Really Happens 107
27 Practical Labs 111
28 Safe Cybersecurity Lab 124

PART IV QUICK REFERENCE 128

29 Command Cheat Sheets 129
30 Dangerous Commands 133
31 Interview Preparation 135

PART V ROADMAPS & MASTER REFERENCE 140

32 Learning Roadmap 141
33 Command Master Reference 146

PART VI APPENDIX 158

34 Glossary 159
35 Further Resources 160

0.1 UNDERSTANDING LINUX

0.1.1 1.1 What Is Linux?

What is it? Linux is a free, open-source **kernel** — the core program that manages a computer's CPU, memory, and devices. It was created by Linus Torvalds and first released in 1991. On its own, a kernel is not something you "use" directly; it needs to be packaged with software (shells, libraries, package managers, desktop environments) to become a usable operating system. That packaged combination is called a **Linux distribution** ("distro").

Why do we need it? Every computer needs software that mediates between raw hardware and the applications you run. Linux fills that role for billions of devices: phones (Android), servers, supercomputers, embedded devices, and desktops.

How does it work? The kernel runs in a privileged CPU mode, controls hardware directly, and exposes a controlled interface (system calls) that user programs use to request services like reading a file or opening a network socket.

> ✓ **TIP**
> Think of the kernel as a car's engine and the distribution as the whole car — engine, body, seats, dashboard. Different manufacturers (distributions) build very different vehicles around a similar engine.

> ◆ **OBJECTIVE**
> Understand the difference between "Linux" (the kernel), "GNU/Linux" (kernel + GNU tools), and a "distribution" (a complete, installable OS built on both).

0.1.2 1.2 A Short History of Linux

- **1969–1970s** — Unix is developed at Bell Labs (Ken Thompson, Dennis Ritchie). Its design — "everything is a file," small composable tools, a hierarchical filesystem — still shapes Linux today.
- **1983** — Richard Stallman starts the **GNU Project**, aiming to build a completely free Unix-compatible operating system. By the late 1980s GNU had produced a compiler (GCC), shell (Bash), and core utilities — but no kernel of its own yet.
- **1991** — Linus Torvalds, a Finnish student, releases the first version of the **Linux kernel** as a hobby project, posting his now-famous "just a hobby, won't be big and professional like gnu" announcement to a newsgroup.
- **Early-mid 1990s** — The Linux kernel is combined with GNU tools to form complete, free operating systems. Distributions like Slackware and Debian emerge.
- **2004** — Ubuntu is released, aiming to make Linux desktop-friendly for newcomers.
- **2000s–today** — Linux becomes dominant in servers, cloud computing, mobile (Android is Linux-based), and embedded systems, while remaining a capable but minority desktop OS.

0.1.3 1.3 The Linux Kernel

The kernel is responsible for:

- **Process management** — creating, scheduling, and terminating running programs
- **Memory management** — allocating RAM, virtual memory, paging
- **Device drivers** — talking to hardware (disks, network cards, GPUs)
- **Filesystem management** — reading/writing files through various filesystem types
- **Networking stack** — implementing TCP/IP and other protocols
- **Security enforcement** — permissions, namespaces, capabilities

Linux uses a **monolithic kernel** design with loadable modules: most core services run in kernel space for performance, but drivers and features can be loaded/unloaded at runtime as **kernel modules** (see Part 24).

0.1.4 1.4 GNU/Linux

The GNU Project supplied the userland tools most distributions still rely on: `bash`, `coreutils` (`ls`, `cp`, `mv`, `rm`…), `gcc`, `glibc`, and more. Because a Linux system is normally "the Linux kernel + GNU tools + other software," many people (including Stallman) argue it should be called **GNU/Linux**. In casual use, "Linux" refers to both the kernel and the whole OS family — context makes the meaning clear.

0.1.5 1.5 What Is a Linux Distribution?

A distribution bundles:

1. The Linux kernel
2. A set of core utilities (usually GNU, sometimes BusyBox on embedded systems)
3. A package manager (APT, DNF, pacman, etc.)
4. Default configuration, an installer, and often a desktop environment

Distributions differ in release philosophy (fixed release vs. rolling release), default software, target audience, and package format. Part 2 covers Ubuntu, Linux Mint, and Kali Linux in detail.

0.1.6 1.6 Open Source vs. Free Software vs. Proprietary Software

Term	Meaning
Proprietary software	Source code is closed; usage is restricted by license; you cannot legally modify or redistribute it.
Open-source software	Source code is publicly available; licenses vary in how much modification/redistribution is allowed.

Term	Meaning
Free software (FSF sense)	"Free as in freedom," not necessarily free of cost. Four freedoms: run, study, modify, and redistribute.

Most Linux distributions combine free/open-source components under licenses like the GPL, MIT, and Apache 2.0.

0.1.7 1.7 Linux vs. Windows

Aspect	Linux	Windows
Source model	Open source (mostly)	Proprietary (mostly closed)
Cost	Free (most distros)	Paid license
Package management	Centralized repositories (APT/DNF/etc.)	Mostly individual installers, increasingly winget/MS Store
Filesystem layout	Single root tree (`/`), no drive letters	Drive letters (`C:\`, `D:\`)
Permissions model	Unix permissions (owner/group/other), root/sudo	ACL-based, Administrator account
Customization	Extremely high (kernel to desktop)	Limited to vendor-exposed options
Common use	Servers, cloud, embedded, developers, security	Consumer desktops, enterprise desktops, gaming
Software installation	Mostly via package manager, dependency-resolved	Mostly via downloaded installers

Neither is universally "better" — they are optimized for different priorities: Windows for broad consumer compatibility and gaming; Linux for control, transparency, and server/infrastructure use.

0.1.8 1.8 Linux vs. macOS

Both are Unix-family systems and share many command-line concepts (macOS is Unix-certified, built on Darwin/BSD roots), which is why most Bash/Unix knowledge transfers directly between Linux and macOS terminals. Key differences: macOS is proprietary, tied to Apple hardware, uses `launchd` instead of `systemd`, and its package management (Homebrew) is third-party rather than built-in.

0.1.9 1.9 Why Developers Use Linux

- Native support for compilers, interpreters, and build tools without emulation layers

- Package managers make installing dev dependencies fast and reproducible
- Most production software (especially backend/cloud) runs on Linux, so developing on Linux reduces "works on my machine" gaps
- Container tooling (Docker) is built directly on Linux kernel features (namespaces, cgroups — see Part 24)
- Full scriptability of the OS itself via Bash

0.1.10 1.10 Why Servers Run Linux

- No per-server licensing cost at scale
- Long-term stability — servers can run for months without reboot
- Fine-grained control over exactly what's installed (minimal attack surface)
- Massive ecosystem of server software (web servers, databases, container orchestration) is Linux-first
- Strong remote administration via SSH

0.1.11 1.11 Why Cybersecurity Professionals Use Linux

- Nearly all security tools (Nmap, Wireshark, Metasploit, and many more) are developed for or run best on Linux
- Purpose-built distributions like Kali Linux ship hundreds of preconfigured security tools
- Deep visibility and control over networking, processes, and the filesystem supports both offensive testing and defensive analysis
- Most real-world targets (servers, cloud infrastructure) are Linux, so understanding Linux internals is directly transferable to assessing them
- Scripting (Bash/Python) integrates natively with the OS for automation

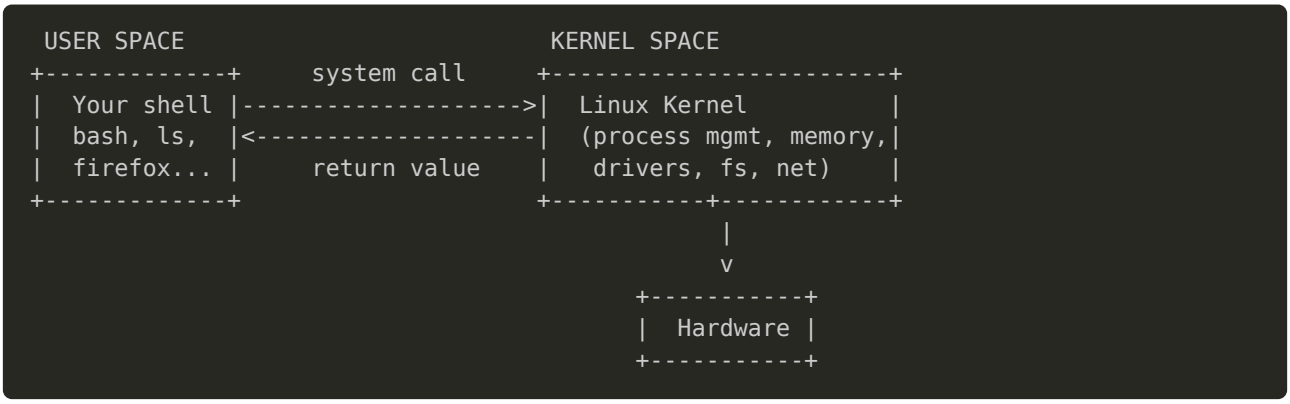
> **SECURITY NOTE**
>
> Owning the operating system a security tool runs on matters. If you don't understand what the OS is doing underneath a tool, you can't fully trust or troubleshoot its output.

0.1.12 1.12 Kernel Space vs. User Space

Modern CPUs support privilege levels. Linux uses this to separate:

- **Kernel space** — where the kernel and drivers run with full hardware access
- **User space** — where normal applications (browsers, editors, your shell) run with restricted access

User-space programs cannot touch hardware directly. Instead they request kernel services through **system calls** (`open()`, `read()`, `write()`, `fork()`, etc.). This separation is a security and stability boundary: a crashing user application normally cannot crash the kernel, and privilege escalation (going from user space to kernel-level control) is treated as a serious security event.



0.1.13 1.13 Shell, Terminal, CLI, GUI

Term	Definition
Terminal (terminal emulator)	A program that provides a text window for typing commands and viewing output (e.g., GNOME Terminal, Konsole, xterm).
Shell	The program that actually interprets and executes the commands you type (e.g., Bash, Zsh, Fish). The terminal displays it; the shell runs it.
CLI	Command-Line Interface — interacting with software purely through typed text commands.
GUI	Graphical User Interface — interacting through windows, icons, and pointer input.

So when you “open a terminal,” you’re really opening a terminal emulator, which then starts a shell (usually Bash) inside it — and that shell is what actually executes the commands you type.

0.1.14 1.14 Desktop Environments

A **desktop environment (DE)** provides the graphical shell: window manager, panel/taskbar, file manager, settings, and bundled apps.

DE	Style	Typical Distros
GNOME	Modern, minimalist, activities-based	Ubuntu (default)
Cinnamon	Traditional Windows-like layout, polished	Linux Mint (default)
MATE	Lightweight, traditional GNOME 2 fork	Linux Mint (alt edition)

DE	Style	Typical Distros
Xfce	Very lightweight, highly configurable	Linux Mint (alt edition), low-resource systems
KDE Plasma	Highly customizable, feature-rich	Kubuntu, many others

A distro can usually run multiple DEs; “Ubuntu” and “GNOME” are separate things that happen to ship together by default.

0.1.15 1.15 Package Managers (Introduction)

A **package manager** installs, updates, configures, and removes software along with its dependencies, using centralized repositories. This solves the “DLL hell” style dependency problems common with manual installers. Ubuntu, Mint, and Kali all use **APT** at their core, since all three are Debian-based. Part 9 covers package management in full depth.

0.1.16 1.16 Linux Filesystem Philosophy

- **“Everything is a file.”** Regular files, directories, devices (`/dev/sda`), and even some kernel interfaces (`/proc`, `/sys`) are represented and accessed through the same file interface.
- **Single hierarchical tree.** There is one root (`/`); external drives are *mounted* into this tree rather than getting separate drive letters like Windows.
- **Small tools, composed together.** Unix philosophy favors small single-purpose programs (`grep`, `sort`, `cut`) combined with pipes rather than giant monolithic applications.
- **Text as a universal interface.** Configuration files and program output are usually plain text, making them easy to inspect, edit, and script.

Part 5 walks through the full filesystem hierarchy (`/etc`, `/var`, `/usr`, etc.) in detail.

0.1.17 1.17 The Full Picture: User → Terminal → Shell → Kernel → Hardware

You type a command → the terminal captures your keystrokes → the shell parses and executes it → the shell (or the program it launched) makes system calls → the kernel performs the privileged operation on real hardware → results flow back up the same chain to your screen.

> ✓ **TIP**
> Whenever a command "does something to the system" — reads a file, sends a packet, changes a permission — somewhere underneath it a system call is being made to the kernel. Keeping this mental model in mind helps you reason about *why* commands behave the way they do, not just *what* to type.

0.1.18 Chapter Summary

- Linux is a kernel; a distribution combines it with GNU tools and other software into a usable OS.
- Linux traces back to Unix design philosophy and the GNU Project, unified by Linus Torvalds's 1991 kernel.
- The kernel/user-space split is a core security and stability boundary enforced via system calls.
- Terminal, shell, CLI, and GUI are distinct but related concepts.
- Desktop environments are swappable graphical shells, independent of the underlying distro.
- "Everything is a file" and hierarchical single-root filesystems are foundational Linux design choices.

0.1.19 Self-Check Questions

1. What is the difference between the Linux kernel and a Linux distribution?
2. Name two reasons servers commonly run Linux instead of Windows.
3. What is a system call, and why does it matter that user-space programs cannot access hardware directly?
4. Is a terminal the same thing as a shell? Explain the difference.
5. Why is macOS command-line knowledge largely transferable to Linux?

0.2 LINUX DISTRIBUTIONS

This chapter looks closely at the three distributions this book focuses on: **Ubuntu**, **Linux Mint**, and **Kali Linux**. All three share a common ancestor — **Debian** — which is why commands and package management stay consistent across all of them.

0.2.1 2.1 Ubuntu

What it is: Ubuntu is a Debian-based distribution developed and commercially sponsored by **Canonical Ltd.**, first released in 2004. Its stated mission is to make Linux accessible to as many people as possible, on desktops, servers, and cloud infrastructure alike.

Ubuntu Desktop vs. Ubuntu Server: - *Ubuntu Desktop* ships with a full graphical environment (GNOME by default) and is aimed at everyday users, developers, and students. - *Ubuntu Server* ships without a GUI by default, is optimized for headless operation (no monitor, accessed via SSH), and is extremely common on cloud platforms (AWS, Azure, Google Cloud all offer official Ubuntu Server images).

LTS releases: Ubuntu releases a new version every six months, but every two years it designates a release as **LTS (Long Term Support)**, guaranteeing five years of security and maintenance updates (extendable further with Ubuntu Pro). LTS releases are named with a year.month scheme (e.g., 22.04, 24.04) and are what almost everyone should install for stability — non-LTS releases are best suited to those who want the newest packages and are comfortable with more frequent upgrade cycles.

> ✓ **TIP**
> When in doubt, install the latest LTS release. It's the version most tutorials, tools, and production servers target.

Package management: Ubuntu uses **APT** (Advanced Package Tool) over `.deb` packages, backed by Canonical's official repositories plus community-maintained PPAs (Personal Package Archives).

Desktop environment: **GNOME** is the default, using an "Activities overview" model rather than a traditional taskbar — a deliberate design choice that can feel unfamiliar to Windows switchers at first.

Best use cases: General desktop use, software development, cloud/server deployments, learning Linux from a well-documented, widely-supported baseline.

Advantages: Huge community and documentation base; excellent hardware support; predictable LTS release cycle; default choice for most cloud providers; strong corporate backing (Canonical).

Disadvantages: GNOME's design takes adjustment for newcomers; some snap-packaged applications (Ubuntu's preferred universal package format) can feel slower to start than native `.deb` equivalents; occasional friction between Canonical's choices (like snaps) and community preferences.

0.2.2 2.2 Linux Mint

What it is: Linux Mint is itself built on top of Ubuntu (which is itself built on Debian), making it "Debian's grandchild." It was created specifically to smooth over rough edges for newcomers, particularly those coming from Windows.

Desktop editions: - **Cinnamon** (Mint's flagship and default) — a modern take on the traditional desktop metaphor: a taskbar, a Start-style menu, and system tray, all of which will feel immediately familiar to a Windows user. - **MATE** — a continuation of the classic GNOME 2 codebase; lighter weight than Cinnamon, extremely stable. - **Xfce** — the lightest of the three, ideal for older or resource-constrained hardware.

Package management: Same as Ubuntu — APT over `.deb` — plus Mint's own **Software Manager** GUI and, notably, Mint tends to avoid pre-installing Snap by default,

preferring traditional `.deb`/Flatpak.

Beginner-friendly features: - Familiar Windows-like layout out of the box (Cinnamon). - Codecs and drivers for common proprietary formats/hardware are easier to enable during setup. - "Update Manager" presents updates with clear risk levels rather than an all-or-nothing approach. - Timeshift (system snapshot tool) is included by default, making it easy to roll back after a bad update.

Best use cases: Everyday desktop use, especially for people switching from Windows; older or lower-spec hardware (via MATE/Xfce); users who want stability over bleeding-edge software.

Advantages: Very approachable learning curve; strong out-of-the-box stability; built-in system snapshot/rollback; inherits Ubuntu's massive software compatibility.

Disadvantages: Smaller team and community than Ubuntu means slightly slower turnaround on some fixes; being downstream of Ubuntu (which is downstream of Debian) means new features can arrive a step later; not a common choice for production servers.

0.2.3 2.3 Kali Linux

What it is: Kali Linux is a Debian-based distribution built and maintained by **Offensive Security**, purpose-built for **penetration testing, security research, and digital forensics**. It's the direct successor to BackTrack Linux.

Why it exists: Instead of making users manually install and configure dozens of separate security tools, Kali ships with hundreds of them pre-installed, pre-configured, and organized by category (recon, exploitation, forensics, wireless, reverse engineering, and more) — turning what would be days of setup into a ready-to-use environment.

Kali vs. Ubuntu:

Aspect	Ubuntu	Kali Linux
Primary purpose	General desktop/server use	Security testing & research
Default user privileges	Standard user, sudo for elevation	Standard user by default since 2020 (older Kali defaulted to root)
Update model	Predictable LTS cycles	Rolling-release, frequent tool updates
Pre-installed software	General productivity apps	Hundreds of security tools
Attack surface	Minimized for general safety	Intentionally includes tools that are also flagged by antivirus/EDR
Recommended daily driver?	Yes	Generally no (see below)

Why Kali should not automatically be your first everyday distro: Kali is optimized for a narrow purpose. It isn't tuned for battery life, general productivity software, or the kind of "just works" daily-use polish that Ubuntu or Mint prioritize. Its toolset (many of which can trigger antivirus alerts, network monitoring alarms, or even legal scrutiny if

misused) also means running it as your everyday personal/work machine increases both your attack surface and the chance of misconfiguration. The recommended pattern — covered fully in Part 28 — is to run Kali inside an **isolated virtual machine**, used deliberately for security work, while your daily-driver OS is Ubuntu, Mint, or something else entirely.

Advantages: Massive, curated toolset ready out of the box; strong documentation from Offensive Security; industry-standard for penetration testing coursework (including OSCP); rolling updates mean current versions of security tools.

Disadvantages: Not beginner-friendly as a first Linux OS; rolling-release model can occasionally introduce instability; carries legal/ethical responsibility — the tools it bundles are powerful and must only be used with proper authorization (see Part 19).

> ❧ **SECURITY NOTE**
> Possessing security tools is not illegal in most jurisdictions, but *using* them against systems or networks you don't own or have explicit written authorization to test can be a serious crime. This book teaches Kali's tools strictly for authorized, legal use: your own systems, local VMs, and platforms that explicitly permit security testing (CTFs, intentionally vulnerable training labs).

0.2.4 2.4 Comparison Table — Ubuntu vs. Mint vs. Kali

Criterion	Ubuntu	Linux Mint	Kali Linux
Difficulty for beginners	Moderate	Easiest	Hardest (not designed for beginners)
Performance (out of box)	Good	Good (better on MATE/Xfce for old hardware)	Good, but resource-heavy tools installed
Stability	Very high (LTS)	Very high	Moderate (rolling release)
Security posture (default)	Strong	Strong	Strong, but larger attack surface by design
Software availability	Excellent	Excellent (same repos as Ubuntu)	Excellent for security tools; general apps less curated
Programming/dev friendliness	Excellent	Excellent	Good, but not its focus
Server usage	Extremely common	Rare	Not intended for server use
Daily desktop usage	Very good	Best of the three	Not recommended as primary daily OS
Cybersecurity learning	Good foundation	Good foundation	Purpose-built

Criterion	Ubuntu	Linux Mint	Kali Linux
Penetration testing	Requires manual tool setup	Requires manual tool setup	Purpose-built, tools pre-installed
Beginner friendliness	Good	Best	Poor (intentionally, given its purpose)

0.2.5 2.5 Choosing the Right Distro for Your Goal

```
Start here
   |
   v
Are you brand new to Linux, coming from Windows?
   |--- Yes --> Linux Mint (Cinnamon)
   |
   No, I want the most widely-documented, industry-standard base
   |--- Yes --> Ubuntu (LTS)
   |
   I specifically need security/pentest tooling AND I already
   have a comfortable daily-driver OS
   |--- Yes --> Kali Linux, inside a VM (see Part 28)
```

For this book's learning path: install **Ubuntu or Mint first** (Parts 3–17 build your general Linux foundation on either), then add **Kali in a VM** once you reach the cybersecurity-focused chapters (Part 18 onward).

0.2.6 Chapter Summary

- Ubuntu, Mint, and Kali are all Debian-based, so commands and package management (`apt`) are consistent across all three.
- Ubuntu balances general-purpose polish with strong documentation and server support.
- Mint prioritizes beginner-friendliness, especially for Windows switchers.
- Kali is a specialized security-testing distro, best run in an isolated VM rather than as a daily driver.
- Choose your starting distro based on your goal, not hype — you can always add others as virtual machines later.

0.3 INSTALLATION

0.3.1 3.1 Installation Methods Overview

There are several ways to get Linux running, and choosing the right one depends on your goals, hardware, and risk tolerance.

Method	Description	Risk of data loss	Best for
VirtualBox / VMware	Run Linux as a "computer inside a computer"	None (isolated virtual disk)	Learning, labs, testing, running Kali
WSL (Windows Subsystem for Linux)	A Linux environment integrated into Windows	None	Developers who need Linux CLI tools inside Windows
Live USB	Boot Linux from a USB stick without installing	None (unless you choose to install from it)	Trying a distro, emergency recovery
Dual boot	Install Linux alongside Windows, choose at startup	Moderate (partitioning Windows disk)	Users who want full native performance but still need Windows
Full/clean installation	Erase existing OS, Linux becomes the only OS	High (erases everything)	Dedicated Linux users, old machines being repurposed
Dedicated Linux computer	A separate physical machine just for Linux	None to your main machine	Serious learners, homelab, security labs

Recommendation for this book: Start with **VirtualBox** or **VMware**. It's free (VirtualBox), completely reversible, safe to experiment on, and is exactly how the labs in this book are designed to be followed — including the Kali/security labs in Part 19 onward, which should almost always be run inside a VM regardless of your experience level.

> ✓ **TIP**
> You do not need to dual-boot or wipe your computer to become proficient in Linux. Every single command, lab, and exercise in this book can be completed inside a virtual machine.

0.3.2 3.2 Bootable USB Creation

If you do want to install Linux directly on hardware (dual boot or full installation), you'll need a bootable USB drive.

ISO files: A distribution is distributed as a single `.iso` file — a complete, byte-for-byte image of an installation disc. Always download ISOs from the **official** site: - Ubuntu: ubuntu.com/download - Linux Mint: linuxmint.com/download.php - Kali Linux: kali.org/get-kali

> ▲ **WARNING**
> Never download ISO files from third-party mirrors, forums, or torrent sites you can't verify. A tampered ISO could contain malware baked directly into your operating

system before you've even logged in for the first time.

Verifying your download (recommended): Official sites publish SHA256 checksums for each ISO. After downloading, compare it against the published value using `sha256sum` (covered fully in Part 20) to confirm the file wasn't corrupted or tampered with.

Tools to create the USB: - **Rufus** (Windows only) — the most common tool for Windows users. 1. Download Rufus from rufus.ie. 2. Insert a USB drive (8GB+ recommended; it will be erased). 3. Open Rufus, select your ISO under "Boot selection." 4. Leave partition scheme as **GPT** for modern UEFI systems (or **MBR** for older BIOS systems — see below). 5. Click **Start** and wait for completion. - **Balena Etcher** (Windows/macOS/Linux) — simpler, cross-platform, fewer options to misconfigure. 1. Download from balena.io/etcher. 2. Select the ISO, select the USB drive, click **Flash**.

USB requirements: At least 8GB capacity (16GB+ safer for Kali, which is larger); USB 3.0 recommended for installation speed.

UEFI vs. BIOS: UEFI is the modern firmware standard (faster boot, supports larger disks, more secure); BIOS (legacy) is older but still found on some machines. Check which your computer uses in its firmware setup screen — most machines made after ~2012 use UEFI.

GPT vs. MBR: GPT (GUID Partition Table) is the modern partitioning scheme paired with UEFI, supports drives over 2TB and more partitions. MBR (Master Boot Record) is the legacy scheme paired with BIOS, limited to 2TB and 4 primary partitions. Use GPT unless your hardware specifically requires MBR/legacy BIOS mode.

Secure Boot: A UEFI security feature that only allows cryptographically signed bootloaders to run. Ubuntu, Mint, and Kali all support Secure Boot out of the box in recent releases — but if you hit boot issues, temporarily disabling Secure Boot in firmware settings is a common troubleshooting step.

Boot menu / boot priority: To boot from your USB instead of the installed OS, you need to either: 1. Press the manufacturer's boot-menu key during startup (commonly F12, F10, F9, or Esc — varies by manufacturer), and select the USB drive, **or** 2. Enter the firmware/BIOS setup (commonly F2 or Del) and reorder boot priority so USB is checked before the internal disk.

0.3.3 3.3 Disk Concepts

- **HDD (Hard Disk Drive)** — spinning magnetic platters; slow, cheap, high capacity.
- **SSD (Solid State Drive)** — flash memory, no moving parts; much faster than HDD.
- **NVMe** — a modern SSD interface connected directly via PCIe, dramatically faster than SATA SSDs.

Partitions divide a physical disk into logical sections. Key partitions in a Linux installation:

- **EFI partition** — a small (typically 100–500MB) FAT32 partition that stores bootloader files UEFI firmware reads directly. Shared between Windows and Linux in dual-boot setups.

- **Root partition (/)** — where the entire Linux filesystem lives (unless you split out separate partitions for `/home`, etc.).
- **`/home`** — often given its own partition so personal files survive a full OS reinstall.
- **`/boot`** — holds the kernel and boot-related files; sometimes separate from `/`, especially with encrypted root partitions.
- **Swap** — disk space used as overflow when RAM is full. Modern systems with plenty of RAM often use a swap *file* rather than a dedicated partition, or a modest swap for hibernation support.

Filesystems: - **ext4** — the default, mature, reliable filesystem for Linux root/home partitions. - **FAT32** — used for the EFI partition and USB drives for maximum compatibility (max 4GB per file — a common gotcha). - **NTFS** — Windows' native filesystem; Linux can read and write NTFS partitions, useful in dual-boot setups for accessing Windows files.

0.3.4 3.4 Dual Boot — Proceed Carefully

> ▲ **WARNING**
> Dual-booting involves resizing an existing Windows partition and creating new ones alongside it. Mistakes here can result in **permanent data loss**. Back up your important files before starting, no exceptions.

Pre-flight checklist:

1. **Back up your data** — an external drive or cloud backup of anything irreplaceable.
2. **Disable BitLocker** (if enabled) — Windows' disk encryption can interfere with partition resizing and, in rare cases, lock you out if the recovery key isn't saved. Suspend or fully decrypt before proceeding, and make sure you have your BitLocker recovery key saved somewhere safe regardless.
3. **Disable Fast Startup** (Windows Settings → Power Options → "Choose what the power buttons do" → uncheck Fast Startup). Fast Startup keeps the Windows filesystem in a "hibernated" state that Linux can't safely read/write, and can cause corruption if left on.
4. **Confirm UEFI mode** — check Windows' System Information app for "BIOS Mode: UEFI" to ensure your Linux installer will match.

Shrinking the Windows partition: 1. Open Windows' **Disk Management** (search "Create and format hard disk partitions"). 2. Right-click your main Windows partition (usually `C:`) → **Shrink Volume**. 3. Enter the amount of space (in MB) to free up for Linux — 40GB+ minimum recommended, more if you plan to store a lot of files. 4. Leave the freed space as **Unallocated** — do not format it yet; the Linux installer will handle that.

Installing (Ubuntu / Mint / Kali all follow this general pattern): 1. Boot from your USB (see 3.2). 2. Choose **"Try"** first if offered, to confirm hardware works, then launch the installer from the desktop. 3. When asked about installation type, choose **"Install alongside Windows Boot Manager"** if offered (simplest, safest) — or **"Something else"** for manual partitioning if you want full control over partition sizes. 4. **Critical step:** if using manual partitioning, triple-check you are creating partitions in the **unal-**

located space you freed up, not on the existing Windows partition. The installer will clearly label each partition with its filesystem and size — read every line before confirming. 5. Confirm the **EFI partition** field points to the *existing* EFI partition (shared with Windows) rather than creating a new one, when given the option. 6. Select the correct **installation disk/bootloader target** — usually the same physical disk Windows is on. 7. Complete the rest of the installer (timezone, user account, password). 8. Reboot; you should now see a **GRUB menu** at startup letting you choose between Linux and Windows.

GRUB is the bootloader most Linux distributions use — the first screen you see after your firmware finishes POST, letting you select which OS (or which kernel version) to boot into.

Common installation mistakes: - Selecting the wrong disk during manual partitioning (always double check disk *and* size, not just the label). - Not disabling Fast Startup, leading to a "dirty" Windows filesystem Linux can't mount properly. - Forgetting to free space before starting the installer. - Choosing "Erase disk" by accident when you meant to dual-boot (this destroys everything on that disk).

> ▲ **WARNING**
> The installer option labeled **"Erase disk and install [distro]"** will destroy *all* data on the target disk, including Windows and all personal files. Only select this if you intend a full, clean installation.

Recovery options if something goes wrong: - **Windows won't boot after installing Linux:** Boot from a Windows installation/recovery USB and use `bootrec /fixmbr` and `bootrec /fixboot` (Windows-side recovery), or use a tool like **Boot-Repair** (Linux-side) to restore GRUB and re-detect Windows. - **GRUB rescue prompt appears:** Usually recoverable by booting from your Linux USB in "Try" mode and running Boot-Repair, or manually reinstalling GRUB from a live session. - **Lost Windows entirely and need it back:** This is why the pre-flight backup step is non-negotiable.

0.3.5 3.5 Post-Install Checklist

Once installed (any method):

1. Connect to Wi-Fi/network.
2. Run a full system update (Part 4).
3. Install any missing drivers (graphics, Wi-Fi chipsets can occasionally need extra steps).
4. Set your timezone and locale correctly if not already configured.
5. Take a VM snapshot (if using a VM) *right now*, before you start experimenting — this gives you a clean baseline to roll back to.

0.3.6 Chapter Summary

- VirtualBox/VMware is the safest, most reversible way to learn Linux — use it for this book's labs.
- Always download ISOs from official sources and verify checksums when possible.

- Dual-booting is safe when you follow the pre-flight checklist (backup, disable BitLocker/Fast Startup) — but mistakes during partitioning are the #1 cause of data loss for beginners.
- GRUB is the bootloader that lets you choose between operating systems at startup.
- Take a VM snapshot immediately after a clean install, before experimenting.

0.4 FIRST DAY WITH LINUX

0.4.1 4.1 Logging In and the Desktop

After installation, you'll reach a login screen asking for your username and password. Once logged in, you'll land on the desktop of whichever desktop environment your distribution uses (GNOME for Ubuntu, Cinnamon for Mint). Take a few minutes to explore: the top or bottom panel/taskbar, the application launcher, and system settings — but don't worry about memorizing the GUI. This book's focus, and where real Linux power lives, is the terminal.

0.4.2 4.2 Opening the Terminal

- **Ubuntu (GNOME):** `Ctrl+Alt+T`, or search "Terminal" in the Activities overview.
- **Linux Mint (Cinnamon):** `Ctrl+Alt+T`, or click the terminal icon in the taskbar.
- **Kali Linux:** Terminal icon is pinned to the taskbar by default; also `Ctrl+Alt+T`.

Useful terminal shortcuts (work across all three):

Shortcut	Effect
`Ctrl+C`	Cancel/interrupt the currently running command
`Ctrl+D`	Signal "end of input" / close the shell
`Ctrl+L`	Clear the screen (same as the `clear` command)
`Ctrl+R`	Search backward through command history
`Tab`	Auto-complete commands, filenames, paths
`Up / Down arrows`	Cycle through previous commands
`Ctrl+A` / `Ctrl+E`	Jump to start / end of the current line
`Ctrl+U`	Clear the line before the cursor

✓ **TIP**

`Tab` completion is one of the highest-leverage habits you can build. Type the first few letters of a file or command and press Tab — the shell will complete it for you, or show you all matching options if you press Tab twice.

0.4.3 4.3 First Commands — Full Reference

For each command below: what it means, syntax, key options, an example, what happens internally, a real-world use case, common mistakes, a safety note where relevant, and a small practice exercise.

`pwd` — Print Working Directory

- **What it means:** Shows the full path of the directory your shell is currently "standing in."
- **Syntax:** `pwd`
- **Example:**

```
$ pwd
/home/student
```

- **Internally:** The shell asks the kernel for the current process's working directory via the `getcwd()` system call.
- **Real-world use:** Confirming where you are before running a command that affects files (especially before something destructive like `rm`).
- **Common mistake:** Assuming you're in a directory because of what a previous `cd` "should" have done — always verify with `pwd` when unsure.
- **Practice:** Open a terminal and run `pwd`. Note the path it prints.

`ls` — List directory contents

- **Syntax:** `ls [options] [path]`
- **Key options:** `-l` (long/detailed listing), `-a` (show hidden files), `-h` (human-readable sizes, use with `-l`), `-lah` (combine all three)
- **Example:**

```
$ ls -lah
total 24K
drwxr-xr-x  5 student student 4.0K Jul 28 09:12 .
drwxr-xr-x  3 root    root    4.0K Jul 20 10:00 ..
-rw-r--r--  1 student student  220 Jul 20 10:00 .bashrc
drwxr-xr-x  2 student student 4.0K Jul 28 09:00 Documents
```

- **Internally:** `ls` calls `opendir()`/`readdir()` to ask the kernel for directory entries, then formats and prints them.
- **Real-world use:** Checking file sizes, permissions, and modification times before copying, editing, or deleting.
- **Common mistake:** Forgetting `-a` and assuming a directory is empty when it actually contains hidden (dotfile) content.
- **Practice:** Run `ls`, then `ls -la`, and compare what's shown.

`cd` — Change Directory

- **Syntax:** `cd [path]`
- **Key shortcuts:** `cd` alone (go to your home directory), `cd ..` (go up one level), `cd -` (go to the previous directory), `cd ~` (home directory)

- **Example:**

```
$ cd Documents
$ pwd
/home/student/Documents
```

- **Internally:** Unlike most commands, `cd` is a **shell builtin**, not a separate program — it must be, because a separate program can't change the working directory of the shell that launched it (a child process cannot alter its parent's state). The shell directly updates its own current-directory pointer.
- **Real-world use:** Navigating your filesystem to reach project folders, config directories, logs, etc.
- **Common mistake:** Typing a path with a typo and getting "No such file or directory" — use Tab completion to avoid this.
- **Practice:** From your home directory, `cd` into `Documents`, then use `cd ..` to go back.

`mkdir` — Make Directory

- **Syntax:** `mkdir [options] directory_name`
- **Key options:** `-p` (create parent directories as needed)
- **Example:**

```
$ mkdir projects
$ mkdir -p projects/linux/labs
```

- **Internally:** Calls the `mkdir()` system call, which asks the kernel's filesystem driver to create a new directory entry.
- **Real-world use:** Organizing files into project folders.
- **Common mistake:** Trying to create a nested path (`projects/linux/labs`) without `-p` when `projects/linux` doesn't exist yet — this fails with an error.
- **Practice:** Create a directory called `practice`, then a nested path `practice/day1/notes` in one command using `-p`.

`rmdir` — Remove (empty) Directory

- **Syntax:** `rmdir directory_name`
- **Example:**

```
$ rmdir practice/day1/notes
```

- **Internally:** Only succeeds if the directory is empty — the kernel refuses to unlink a directory entry that still has contents.
- **Real-world use:** Cleaning up empty folders left over from a project.
- **Common mistake:** Trying to `rmdir` a non-empty directory (it will fail — you'd need `rm -r` instead, covered below with its warning).

- **Practice:** Try `rmdir` on an empty folder, then try it on one that still has a file inside and observe the error.

`touch` — Create an empty file (or update its timestamp)

- **Syntax:** `touch filename`
- **Example:**

```
$ touch notes.txt
```

- **Internally:** If the file doesn't exist, `touch` creates an empty one via the `open()` system call with a "create" flag. If it exists, it just updates the file's modification timestamp.
- **Real-world use:** Quickly creating placeholder files, or "bumping" a file's timestamp to trigger a build system that watches for changes.
- **Common mistake:** Assuming `touch` edits file content — it doesn't; it only creates or "touches" the timestamp.
- **Practice:** Create a file called `todo.txt` with `touch`, then run `ls -l` to see its size (0 bytes) and timestamp.

`cp` — Copy files/directories

- **Syntax:** `cp [options] source destination`
- **Key options:** `-r` (recursive, required for directories), `-v` (verbose, show what's being copied), `-i` (prompt before overwriting)
- **Example:**

```
$ cp notes.txt notes_backup.txt
$ cp -r projects/ projects_backup/
```

- **Internally:** Opens the source file, reads its contents, and writes a new file at the destination — a genuine duplication of data, not a reference.
- **Real-world use:** Backing up a file before editing it.
- **Common mistake:** Forgetting `-r` when copying a directory (you'll get an "omitting directory" error).
- **Safety note:** `cp` will silently overwrite an existing destination file unless you use `-i` to be prompted first.
- **Practice:** Copy `notes.txt` to `notes_copy.txt` and confirm both exist with `ls`.

`mv` — Move or rename files/directories

- **Syntax:** `mv [options] source destination`
- **Example:**

```
$ mv notes.txt renamed_notes.txt      # rename
$ mv renamed_notes.txt Documents/     # move into a folder
```

- **Internally:** If source and destination are on the same filesystem, `mv` simply updates the directory entry (near-instant, no data copied). Across different filesystems/partitions, it performs a copy-then-delete internally.
- **Real-world use:** Renaming files, reorganizing folders.
- **Common mistake:** Moving a file into a destination that doesn't exist as a directory — it will instead be renamed *to* that non-existent name rather than moved into it, which can be surprising.
- **Practice:** Rename `todo.txt` to `today.txt`, then move it into `Documents/`.

`rm` — Remove files

- **Syntax:** `rm [options] file`
- **Key options:** `-r` (recursive, for directories), `-f` (force, no confirmation), `-i` (prompt before every deletion)
- **Example:**

```
$ rm old_notes.txt
$ rm -r old_project/
```

> ▲ **WARNING**
> `rm` does **not** move files to a Recycle Bin/Trash — deletion via the terminal is immediate and, in almost all cases, unrecoverable. `rm -rf` (recursive + force) is one of the most dangerous commands you will learn in this book. Full treatment, including the infamous `rm -rf /` scenario, is in Part 30.

- **Internally:** Calls `unlink()` to remove the directory entry pointing to the file's data. The underlying data isn't necessarily wiped immediately, but there is no built-in "undo" — recovery (if possible at all) requires specialized forensic tools and is not guaranteed.
- **Real-world use:** Cleaning up temporary files, deleting old backups.
- **Common mistake:** Using wildcards carelessly, e.g. `rm *` in the wrong directory, deleting far more than intended.
- **Practice:** Create a throwaway file with `touch delete_me.txt`, confirm it exists with `ls`, then remove it with `rm delete_me.txt` and confirm it's gone.

`cat` — Concatenate and display file contents

- **Syntax:** `cat filename`
- **Example:**

```
$ cat notes.txt
Meeting at 3pm.
```

- **Internally:** Reads the file's contents via `read()` system calls and writes them straight to standard output.
- **Real-world use:** Quickly viewing short files, or combining multiple files (`cat file1 file2 > combined.txt`).
- **Common mistake:** Using `cat` on a very large file, flooding your terminal — for large files, use `less` instead.
- **Practice:** Run `cat` on any text file you've created so far.

`less` — View file contents one page at a time

- **Syntax:** `less filename`
- **Key keys inside less:** `Space`/`f` (next page), `b` (previous page), `/searchterm` (search forward), `q` (quit)
- **Example:**

```
$ less /var/log/syslog
```

- **Internally:** Unlike `cat`, `less` doesn't load the whole file into memory or dump it all to the screen at once — it reads and displays content on demand as you scroll, which is why it handles huge files comfortably.
- **Real-world use:** Reading long log files or documentation.
- **Common mistake:** Forgetting `q` quits — new users sometimes think the terminal has "frozen" while inside `less`.
- **Practice:** Open a long file (or `man ls`) with `less` and practice scrolling and searching with `/`.

`clear` — Clear the terminal screen

- **Syntax:** `clear`
- **Example:** `$ clear`
- **Internally:** Sends a terminal control sequence that instructs the terminal emulator to wipe visible content — it doesn't affect command history or running processes at all.
- **Real-world use:** Decluttering your view before starting a new task.
- **Practice:** Run a few commands, then `clear` and confirm the screen is wiped.

`history` — Show previously run commands

- **Syntax:** `history`
- **Example:**

```
$ history
  501  pwd
  502  ls -la
  503  cd Documents
```

- **Internally:** Bash keeps an in-memory list of commands during your session and persists it to `~/.bash_history` on exit.
- **Real-world use:** Recalling a complex command you ran earlier instead of retyping it (`!502` re-runs history entry 502).
- **Security note:** Command history can retain sensitive information (like a password typed by mistake on the command line) — be mindful of what ends up in `~/.bash_history`.
- **Practice:** Run `history` after using a few commands and identify the entry number of your last `cd`.

`whoami` — Show the current logged-in username

- **Syntax:** `whoami`
- **Example:**

```
$ whoami
student
```

- **Real-world use:** Confirming which user context a script or session is running under — especially important before running commands with `sudo`.
- **Practice:** Run `whoami` and compare it to the username shown in your terminal prompt.

`date` — Show or set the system date/time

- **Syntax:** `date [+FORMAT]`
- **Example:**

```
$ date
Tue Jul 28 09:14:22 UTC 2026
$ date +"%Y-%m-%d"
2026-07-28
```

- **Real-world use:** Timestamping logs and backup filenames in scripts.
- **Practice:** Run `date`, then try `date +"%H:%M:%S"` to show just the time.

`cal` — Display a calendar

- **Syntax:** `cal [month] [year]`
- **Example:**

```
$ cal
$ cal 12 2026
```

- **Real-world use:** Rare in daily admin work, but handy for quick reference in scripts/terminals.
- **Practice:** Run `cal` to see the current month.

`uname` — Print system/kernel information

- **Syntax:** `uname [options]`
- **Key options:** `-a` (all information), `-r` (kernel release version)
- **Example:**

```
$ uname -a
Linux ubuntu-vm 6.8.0-generic #1 SMP x86_64 GNU/Linux
```

- **Real-world use:** Confirming kernel version before installing kernel-specific software or drivers.
- **Practice:** Run `uname -a` on your system and identify the kernel version.

`hostname` — Show or set the system's network hostname

- **Syntax:** `hostname`
- **Example:**

```
$ hostname
ubuntu-vm
```

- **Real-world use:** Identifying which machine you're on when managing multiple servers over SSH.
- **Practice:** Run `hostname` and note the result.

`reboot` — Restart the system

- **Syntax:** `sudo reboot` ::: warning Requires root privileges. Always save your work before running this — open applications will be closed immediately. :::
- **Practice:** (Optional — only if you're ready to restart) `sudo reboot`.

`shutdown` — Power off (or schedule shutdown of) the system

- **Syntax:** `sudo shutdown [options] [time]`
- **Key options:** `-h now` (halt immediately), `-r now` (restart immediately), `+10` (shutdown in 10 minutes)
- **Example:**

```
$ sudo shutdown -h now
$ sudo shutdown -r +5      # restart in 5 minutes
```

❧ **SECURITY NOTE**

Both `reboot` and `shutdown` require elevated (root/sudo) privileges precisely because they affect every user and process on the system — a safety measure against accidental or malicious disruption by an unprivileged user.

0.4.4 4.4 Updating and Installing Software (Preview)

You'll cover package management fully in Part 9, but as a first-day essential:

```
sudo apt update       # refresh the list of available packages
sudo apt upgrade      # install available updates
sudo apt install <package_name>   # install new software
sudo apt remove <package_name>    # remove software
```

✓ **TIP**

Run `sudo apt update && sudo apt upgrade` shortly after installing Linux, and periodically afterward, to keep your system secure and current.

0.4.5 4.5 Checking System Info, Wi-Fi, Bluetooth, and Displays

- **System info (GUI):** Settings → About (shows OS version, hardware summary).
- **Wi-Fi:** Click the network icon in your system tray/panel, select your network, enter the password. On the command line, `nmcli` (covered in Part 12) can do this without the GUI.
- **Bluetooth:** Settings → Bluetooth, toggle on, pair devices as you would on any OS.
- **Display settings:** Settings → Displays, for resolution, scaling, and multi-monitor arrangement.

0.4.6 4.6 File Manager and Creating Files/Folders (GUI)

Each distro ships a graphical file manager (Nautilus on Ubuntu/GNOME, Nemo on Mint/Cinnamon) that lets you create folders (right-click → New Folder), rename, copy, and delete files just as you would on Windows or macOS. It's worth knowing it exists, but as you progress through this book, you'll find the terminal is faster and more powerful for almost everything.

0.4.7 4.7 First-Day Practice Set

1. Open a terminal and confirm your current directory with `pwd`.
2. Create a folder called `linux_practice`.
3. Move into it with `cd`.
4. Create three files: `notes.txt`, `todo.txt`, `ideas.txt` using `touch`.

5. List them with `ls -l`.
6. Copy `notes.txt` to `notes_backup.txt`.
7. Rename `ideas.txt` to `important_ideas.txt` using `mv`.
8. View the contents of `todo.txt` with `cat` (it will be empty — that's expected).
9. Delete `notes_backup.txt` with `rm`.
10. Run `history` and find the command where you created your practice folder.

0.4.8 Chapter Summary

- The terminal + shell combination is where real Linux productivity happens.
- Twenty foundational commands — `pwd`, `ls`, `cd`, `mkdir`, `rmdir`, `touch`, `cp`, `mv`, `rm`, `cat`, `less`, `clear`, `history`, `whoami`, `date`, `cal`, `uname`, `hostname`, `reboot`, `shutdown` — cover the vast majority of everyday navigation and file operations.
- `rm` is permanent and does not use a Recycle Bin — always double-check before running it, especially with wildcards.
- `sudo apt update && sudo apt upgrade` should become a regular habit.

0.5 THE LINUX FILESYSTEM

0.5.1 5.1 The Filesystem Hierarchy Standard (FHS)

Every Linux distribution organizes its files according to a shared convention called the **Filesystem Hierarchy Standard**. Unlike Windows, there is exactly one root, written `/`, and every disk, partition, or device is *mounted* somewhere inside that single tree.

```
/
├── bin -> usr/bin      (essential user commands)
├── sbin -> usr/sbin    (essential system admin commands)
├── boot                (kernel, bootloader files)
├── dev                 (device files)
├── etc                 (system-wide configuration)
├── home                (personal user directories)
├── lib -> usr/lib       (shared libraries)
├── media                (auto-mounted removable media)
├── mnt                  (temporary manual mount point)
├── opt                  (optional/third-party software)
├── proc                 (virtual: running process info)
├── root                 (home directory for the root user)
├── sys                  (virtual: kernel/device info)
├── tmp                  (temporary files, cleared on reboot)
├── usr                  (user programs, libraries, docs)
└── var                  (variable data: logs, spool, caches)
```

0.5.2 5.2 Directory-by-Directory Breakdown

- `/` — the root of the entire filesystem tree. Everything descends from here.

- **`/home`** — each regular user gets a subdirectory here (e.g., `/home/student`) for personal files, configuration, and documents.
- **`/root`** — the home directory for the **root** (administrator) user — separate from `/`, and not to be confused with it.
- **`/etc`** — nearly all system-wide configuration files live here in plain text (network settings, user account definitions, service configs).
- **`/var`** — "variable" data that changes as the system runs: logs (`/var/log`), package caches, mail spools, web server files.
- **`/usr`** — the bulk of installed software, libraries, and documentation lives here — despite the name, it's not really about the "user," but historically stood for "Unix System Resources."
- **`/bin`, `/sbin`** — essential command binaries (on modern systems, these are usually symlinks into `/usr/bin` and `/usr/sbin`).
- **`/tmp`** — scratch space for temporary files; typically wiped on every reboot.
- **`/opt`** — a convention for self-contained, optional third-party software packages.
- **`/dev`** — special files representing hardware devices (e.g., `/dev/sda` for a disk, `/dev/null` for a "black hole" you can write to) — a direct expression of "everything is a file."
- **`/proc`** — a **virtual filesystem**, generated live by the kernel, exposing information about running processes and kernel parameters (e.g., `/proc/cpuinfo`, `/proc/1234/status`). Nothing here exists on disk; it's created on demand.
- **`/sys`** — another virtual filesystem, exposing kernel and device/driver information in a more structured way than `/proc`.
- **`/mnt`, `/media`** — conventional mount points: `/media` is typically used for auto-mounted removable devices (USB drives), `/mnt` for temporary manual mounts by an administrator.
- **`/boot`** — the Linux kernel image, initial RAM disk (initramfs), and bootloader configuration.

0.5.3 5.3 Why This Structure Exists

The FHS exists so that software, scripts, and administrators can rely on consistent, predictable locations across *any* Linux distribution — a script that reads logs from `/var/log` or configuration from `/etc` will work the same way on Ubuntu, Mint, Kali, or any other FHS-compliant distro. It also separates data by *purpose and volatility*: configuration (`/etc`) is separated from user data (`/home`), which is separated from constantly-changing runtime data (`/var`), which is separated from installed software (`/usr`). This separation makes backups, permissions, and even separate disk partitions for different purposes much simpler to manage in production.

0.5.4 5.4 Filesystem Navigation & Inspection Commands

`tree` — Display a directory structure as a visual tree.

```
$ tree Documents
Documents
├── notes.txt
└── projects
```

```
    └── linux
```

(If not installed: `sudo apt install tree`.)

`find` — Search for files by name, type, size, modification time, and more; searches recursively.

```
$ find /home/student -name "*.txt"
$ find . -type d -name "backup*"
$ find /var/log -mtime -1        # modified in the last 1 day
```

`locate` — Search a pre-built index of filenames (much faster than `find`, but can be slightly out of date until the index refreshes).

```
$ locate nginx.conf
```

(May need `sudo apt install mlocate` and an initial `sudo updatedb`.)

`which` — Show the full path of the executable that would run for a given command name (searches your `PATH`).

```
$ which python3
/usr/bin/python3
```

`whereis` — Similar to `which`, but also shows the binary, source, and manual page locations.

```
$ whereis ls
ls: /usr/bin/ls /usr/share/man/man1/ls.1.gz
```

`file` — Identify a file's type by inspecting its content (not just its extension).

```
$ file report.pdf
report.pdf: PDF document, version 1.7
```

`stat` — Show detailed metadata about a file: size, permissions, timestamps, inode number.

```
$ stat notes.txt
  File: notes.txt
  Size: 220        Blocks: 8
Access: (0644/-rw-r--r--)  Uid: (1000/student)
```

`du` — Show disk usage of files/directories.

```
$ du -sh Documents/     # -s summary, -h human-readable
120M    Documents/
```

`df` — Show disk space usage for entire mounted filesystems.

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        50G   12G   36G  26% /
```

✓ **TIP**

`du` tells you how much space *specific files/folders* use. `df` tells you how full an *entire disk/partition* is. They answer different questions and are often used together when troubleshooting "disk full" errors (see Part 16).

0.5.5 5.5 Practical Exercises

1. Run `tree /etc` (or `ls /etc` if `tree` isn't installed) and identify three configuration files you recognize.
2. Use `find` to locate all `.log` files under `/var/log`.
3. Run `df -h` and identify how full your root partition (`/`) is.
4. Use `stat` on a file you created in Part 4 and identify its exact permission bits.
5. Run `which bash` and `whereis bash` and compare their output.

0.5.6 Chapter Summary

- Linux uses one unified tree rooted at `/`; there are no drive letters.
- Each top-level directory has a specific, standardized purpose (config in `/etc`, logs in `/var`, personal files in `/home`, and so on).
- `/proc` and `/sys` are virtual filesystems generated live by the kernel — a direct window into running system state.
- `find`/`locate` search for files; `which`/`whereis` locate commands; `du`/`df` measure disk usage at different scopes.

0.6 FILE MANAGEMENT

0.6.1 6.1 Paths: Absolute vs. Relative

An **absolute path** starts from the root `/` and fully specifies a location regardless of where you currently are: `/home/student/Documents/notes.txt`. A **relative path** is interpreted starting from your current directory: if you're in `/home/student`, then `Documents/notes.txt` refers to the same file.

- `.` — the current directory.
- `..` — the parent directory (one level up).
- `~` — shorthand for your home directory (e.g., `/home/student`).

```
$ cd ~/Documents      # go to Documents inside your home directory
$ cd ..               # go up one level
$ cp ./notes.txt ../backup/   # copy from current dir to sibling dir
```

0.6.2 6.2 Wildcards (Globbing)

The shell expands certain special characters into matching filenames *before* running a command:

Wildcard	Matches
`*`	Any number of characters (including zero)
`?`	Exactly one character
`[abc]`	Any one character in the set `a`, `b`, or `c`
`[a-z]`	Any one character in the range a through z

```
$ ls *.txt          # all files ending in .txt
$ ls report?.csv    # report1.csv, report2.csv, but not report10.csv
$ ls [A-C]*         # files starting with A, B, or C
```

> ▲ **WARNING**
> Wildcards are expanded by the *shell*, not by the command itself, and they expand to **whatever currently matches** — `rm *.tmp` in the wrong directory can delete far more than you expect. Always run `ls` with the same pattern first to preview what will be affected before using it with `rm`, `mv`, or `cp`.

0.6.3 6.3 Hidden Files

Any file or directory whose name starts with a dot (`.`) is treated as **hidden** by convention — it won't show in a plain `ls` or default file-manager view. This is used heavily for configuration files (`.bashrc`, `.ssh/`, `.gitconfig`) to keep them out of the way during normal browsing. Use `ls -a` to reveal them.

0.6.4 6.4 Core File Management Commands

Command	Purpose	Example
`touch`	Create empty file / update timestamp	`touch file.txt`
`mkdir`	Create directory (`-p` for nested)	`mkdir -p a/b/c`
`cp`	Copy (`-r` for directories)	`cp -r src/ dst/`
`mv`	Move or rename	`mv old.txt new.txt`
`rm`	Delete (`-r` recursive, `-f` force)	`rm -r old_folder/`
`rmdir`	Delete an *empty* directory only	`rmdir empty_folder`
`ln`	Create links (see 6.5)	`ln -s target linkname`

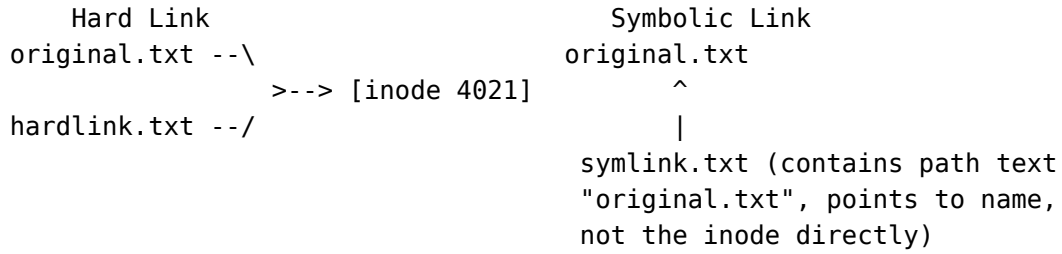
(Full per-command breakdowns for touch/mkdir/cp/mv/rm/rmdir were covered in Part 4; this section builds on them.)

0.6.5 6.5 Symbolic Links vs. Hard Links

Every file on an ext4 filesystem is represented internally by an **inode** — a data structure holding its metadata (permissions, owner, size, and pointers to the actual data blocks on disk). A filename is really just a label pointing to an inode.

- **Hard link** — a second filename pointing to the *same inode* as the original. Both names are equally "real"; deleting one leaves the data accessible through the other (the data is only actually freed once *all* hard links to it are removed). Hard links cannot cross filesystems/partitions, and cannot point to directories.
- **Symbolic link (symlink)** — a small separate file that simply contains the *path* to another file, similar to a Windows shortcut. If the original file is deleted or moved, the symlink breaks ("dangling link"). Symlinks can cross filesystems and can point to directories.

```
$ ln original.txt hardlink.txt        # hard link
$ ln -s original.txt symlink.txt      # symbolic link
$ ls -l symlink.txt
lrwxrwxrwx 1 student student 12 Jul 28 09:20 symlink.txt -> original.txt
```



0.6.6 6.6 Archiving and Compression

`tar` — bundle multiple files/directories into a single archive (does not compress by itself unless combined with a compression flag).

```
$ tar -cvf archive.tar Documents/         # create
$ tar -xvf archive.tar                    # extract
$ tar -czvf archive.tar.gz Documents/     # create + gzip compress
$ tar -xzvf archive.tar.gz                # extract a .tar.gz
```

Flags: `-c` create, `-x` extract, `-v` verbose, `-f` specify filename, `-z` gzip compression.

`gzip` / `gunzip` — compress/decompress a single file (replaces the original by default).

```
$ gzip notes.txt        # produces notes.txt.gz, removes notes.txt
$ gunzip notes.txt.gz   # restores notes.txt
```

`zip` / `unzip` — create/extract `.zip` archives, the most cross-platform-friendly format (works natively on Windows/macOS too).

```
$ zip -r archive.zip Documents/
$ unzip archive.zip
```

✓ **TIP**

Use `tar.gz` for Linux-to-Linux transfers (it preserves Unix permissions and ownership cleanly) and `.zip` when sharing archives with Windows or macOS users.

0.6.7 6.7 Practical Exercises

1. Create a directory tree `project/src`, `project/docs`, `project/tests` in a single `mkdir -p` invocation… actually using multiple paths: `mkdir -p project/{src,docs,tests}`.
2. Create a file inside `project/src`, then create a symbolic link to it in your home directory.
3. Create a hard link to the same file and confirm (`ls -li`) that both share the same inode number.
4. Archive the entire `project/` directory into `project.tar.gz`.
5. Delete the original `project/` folder, then extract `project.tar.gz` to restore it.
6. Use a wildcard to list only `.txt` files in a folder containing mixed file types.
7. Create a hidden file called `.secrets.txt` and confirm it doesn't appear in a plain `ls`.
8. Use `rmdir` on a non-empty directory and observe the error message it gives you.

0.6.8 Chapter Summary

- Absolute paths always start with `/`; relative paths depend on your current directory.
- Wildcards are expanded by the shell before the command runs — always preview with `ls` before using them with destructive commands.
- Hard links share an inode with the original file; symbolic links are separate files pointing to a path and can break if the target moves.
- `tar` bundles files (optionally with compression); `zip`/`unzip` is the most cross-platform-friendly archive format.

0.7 READING AND EDITING FILES

0.7.1 7.1 Quick-Viewing Commands

Command	Purpose	Notes
`cat`	Dump entire file to screen	Best for short files
`less`	Page through a file interactively	Best for long files/logs; doesn't load whole file into memory
`more`	Older pager, page-forward only	Mostly superseded by `less`
`head`	Show the first N lines (default 10)	`head -20 file.log`

Command	Purpose	Notes
`tail`	Show the last N lines (default 10)	`tail -f file.log` follows new lines live

```
$ head -5 access.log
$ tail -f /var/log/syslog     # watch a log file update in real time — Ctrl+C to stop
```

✓ **TIP**

`tail -f` is one of the most-used commands in real-world Linux administration — it lets you watch a log file as new events are written, essential for live troubleshooting (covered further in Part 16).

0.7.2 7.2 nano — The Beginner-Friendly Editor

`nano` is a simple terminal text editor with on-screen shortcut hints, making it a comfortable starting point.

```
$ nano notes.txt
```

Inside nano: type normally to edit. `Ctrl+O` saves ("Write Out"), `Ctrl+X` exits, `Ctrl+K` cuts a line, `Ctrl+U` pastes it back, `Ctrl+W` searches. The bottom of the screen always shows the available shortcuts (`^` represents Ctrl).

0.7.3 7.3 Vim Fundamentals

`vim` (an improved version of the classic `vi`) is far more powerful than nano once learned, and is installed by default (or trivially available) on nearly every Linux system you will ever touch — including minimal servers where nano might not be. Investing time in Vim basics pays off across your entire career.

Vim is **modal** — the same keys do different things depending on which mode you're in.

- **Normal mode** (default on open) — keys are commands, not text input. Navigation and editing operations happen here.
- **Insert mode** — press `i` to enter; typing inserts text as in a normal editor. Press `Esc` to return to Normal mode.
- **Visual mode** — press `v` to enter; lets you select text (like holding Shift and arrow keys) for operations like copy/delete.
- **Command mode** — press `:` from Normal mode to type commands like save, quit, or search-and-replace.

Saving and exiting (typed after `:`): | Command | Effect | |—|—| | `:w` | Save (write) | | `:q` | Quit (fails if there are unsaved changes) | | `:wq` or `:x` | Save and quit | | `:q!` | Quit **without** saving, discarding changes |

Navigation (Normal mode): | Key | Moves | |—|—| | `h j k l` | left / down / up / right | | `w` | forward one word | | `b` | backward one word | | `0` | start of line | | `$` | end of line | | `gg` | start of file | | `G` | end of file | | `:42` | jump to line 42 |

Searching: | Key | Effect | |—|—| | `/pattern` | search forward for “pattern” | | `?pattern` | search backward | | `n` | repeat search, same direction | | `N` | repeat search, opposite direction |

Copy/paste (Normal mode): | Key | Effect | |—|—| | `yy` | yank (copy) the current line | | `dd` | delete (cut) the current line | | `p` | paste after the cursor | | `x` | delete a single character | | `u` | undo | | `Ctrl+r` | redo |

0.7.4 7.4 Vim Cheat Sheet

```
MODES               NAVIGATION          EDITING              SAVE/QUIT
i  insert before     h j k l  ← ↓ ↑ →    x    delete char      :w    save
a  insert after      w / b    word ± 1   dd   delete line      :q    quit
Esc  → normal        0 / $    line ± end yy   yank (copy) line :wq   save+quit
v  visual mode       gg / G   file ± end p    paste            :q!   discard+quit
:  command mode      /text    search      u    undo
                     n / N    repeat ±    Ctrl+r  redo
```

> ★ **LAB**
> **Objective:** Get comfortable with basic Vim editing.
> **Steps:** 1. Run `vim practice.txt`. 2. Press `i`, type three lines of text, press `Esc`. 3. Move to the first line with `gg`. 4. Yank it with `yy`, move to the last line with `G`, paste with `p`. 5. Search for a word you typed using `/word`. 6. Save and quit with `:wq`. 7. Reopen the file with `cat practice.txt` to confirm your edits persisted.

0.7.5 7.5 Exercises

1. Create a file with `nano`, write a short paragraph, save and exit.
2. Open the same file in `vim`, add a new line at the end, and save.
3. Use `head -1` and `tail -1` on the file to confirm the first and last lines.
4. Practice `dd`/`p` in Vim to reorder two lines.
5. Open a large file (e.g., `/var/log/dpkg.log` if present) in `less` and search for a term with `/`.

0.7.6 Chapter Summary

- `cat` for short files, `less` for anything long, `head`/`tail` for previewing beginnings/ends of files.
- `nano` is beginner-friendly; `vim` is more powerful and near-universally available, making it worth learning properly.
- Vim’s modal design (Normal/Insert/Visual/Command) takes a bit of practice but becomes extremely fast once internalized.

0.8 USERS, GROUPS AND PERMISSIONS

0.8.1 8.1 Users and Groups

Linux is a genuinely **multi-user** operating system at its core — every process runs as a specific user, and every file is owned by a specific user and group. This isn't a bolted-on feature; it's foundational to how permissions and security work throughout the entire OS.

- **User** — an account identified internally by a numeric **UID** (User ID). UID 0 is always **root**, the administrator account with unrestricted access.
- **Group** — a collection of users, identified by a numeric **GID** (Group ID), used to share permissions among multiple people without granting them individually.
- **root** — the superuser. Root can read, write, and execute any file, and bypass normal permission checks entirely. Because of this, working *as* root for everyday tasks is discouraged — a single mistake (or compromised process) has unrestricted power.

Key files: - **`/etc/passwd`** — one line per user account: username, UID, GID, home directory, default shell. (Despite the name, it no longer stores actual passwords — that ended decades ago for security reasons.)

```
student:x:1000:1000:Student User:/home/student:/bin/bash
```

- **`/etc/group`** — one line per group, listing its GID and member usernames.
- **`/etc/shadow`** — stores actual **hashed** passwords and password-aging policy, readable only by root. Separating this from `/etc/passwd` (which must be world-readable for many system functions) is a deliberate security measure.

0.8.2 8.2 sudo — Controlled Privilege Escalation

Rather than logging in as root directly, Linux's modern convention is for a trusted regular user to run individual commands *as* root using `sudo` ("superuser do"), after re-entering their own password.

```
$ sudo apt update
[sudo] password for student:
```

SECURITY NOTE

`sudo` is safer than a root login because: (1) it requires re-authentication, (2) it can log exactly which commands were escalated and by whom (in `/var/log/auth.log`), and (3) it can be scoped — an administrator can grant a user permission to run only *specific* commands as root, rather than full root access. Never share your sudo password, and be deliberate about exactly which commands you prefix with `sudo`.

0.8.3 8.3 User & Group Management Commands

Command	Purpose	Example
`whoami`	Show current username	`whoami`
`id`	Show UID, GID, and group memberships	`id student`
`who`	Show who is currently logged in	`who`
`w`	Like `who`, plus what each user is doing	`w`
`users`	List logged-in usernames only	`users`
`useradd`	Create a user (low-level; often used with options)	`sudo useradd -m newuser`
`adduser`	Create a user (friendlier, interactive, Debian-family)	`sudo adduser newuser`
`usermod`	Modify an existing user (e.g., add to a group)	`sudo usermod -aG sudo newuser`
`userdel`	Delete a user	`sudo userdel -r newuser`
`groupadd`	Create a new group	`sudo groupadd developers`
`groups`	Show groups a user belongs to	`groups student`
`passwd`	Change a password	`passwd` (own) / `sudo passwd newuser`
`su`	Switch to another user (or root)	`su - newuser`
`sudo`	Run a single command as root (or another user)	`sudo command`

```
$ sudo adduser alice
$ sudo usermod -aG sudo alice     # grant alice sudo privileges
$ id alice
uid=1001(alice) gid=1001(alice) groups=1001(alice),27(sudo)
```

✓ **TIP**

On Ubuntu/Mint/Kali, membership in the `sudo` group is what grants sudo privileges — adding a user to that group (`usermod -aG sudo username`) is the standard way to make them an administrator.

0.8.4 8.4 Permission Model

Every file and directory has three permission categories, each with three possible permission bits:

Category	Meaning
Owner (user)	The user who owns the file

Category	Meaning
Group	The group that owns the file
Other	Everyone else

Bit	Meaning on a **file**	Meaning on a **directory**
`r` (read)	View file contents	List directory contents
`w` (write)	Modify file contents	Create/delete/rename entries inside
`x` (execute)	Run the file as a program/script	Enter (`cd` into) the directory

```
$ ls -l notes.txt
-rw-r--r-- 1 student student 220 Jul 28 09:14 notes.txt
```

Reading `-rw-r--r--` left to right: first character is the file type (`-` regular file, `d` directory, `l` symlink), then three groups of three: owner (`rw-`), group (`r--`), other (`r--`).

0.8.5 8.5 chmod, chown, chgrp

`chmod` — change permissions.

```
$ chmod u+x script.sh        # add execute for owner
$ chmod g-w file.txt         # remove write for group
$ chmod 755 script.sh        # set exact permissions numerically (below)
```

`chown` — change file owner (and optionally group).

```
$ sudo chown alice notes.txt
$ sudo chown alice:developers notes.txt   # owner and group together
```

`chgrp` — change just the group.

```
$ sudo chgrp developers notes.txt
```

`umask` — sets the *default* permissions removed from newly created files. A umask of `022` means new files get `644` (rw-r–r–) and new directories get `755` by default, rather than the theoretical maximum of `666`/`777`.

0.8.6 8.6 Numeric (Octal) Permissions

Each permission bit has a value: `r=4`, `w=2`, `x=1`. Add them per category to get one digit per owner/group/other.

```
      Owner    Group    Other
r w x    r w x    r w x
4 2 1    4 2 1    4 2 1

755  =  rwx  r-x  r-x    (7=4+2+1, 5=4+1, 5=4+1)
644  =  rw-  r--  r--    (6=4+2,   4=4,   4=4)
700  =  rwx  ---  ---    (owner only, fully)
600  =  rw-  ---  ---    (owner can read/write only, no execute, nobody else)
```

Worked example: you want a script only *you* can read, write, and execute, and no one else can touch it at all:

```
Owner: read+write+execute = 4+2+1 = 7
Group: nothing             = 0
Other: nothing              = 0
→ chmod 700 script.sh
```

Worked example: a shared team document — you can edit, your group can edit, everyone else can only read:

```
Owner: rw-  = 4+2   = 6
Group: rw-  = 4+2   = 6
Other: r--  = 4     = 4
→ chmod 664 shared_doc.txt
```

Numeric	Symbolic	Typical use
777	rwxrwxrwx	Rarely appropriate — full access for everyone
755	rwxr-xr-x	Scripts/programs everyone can run, only owner can edit
700	rwx——	Private scripts/directories only you should touch
644	rw-r–r–	Standard files: you can edit, everyone can read
600	rw——-	Sensitive files (SSH private keys, credentials)

SECURITY NOTE

Setting 777 on a file or directory means *anyone* on the system can modify it — including, in some scenarios, other unprivileged users or compromised processes. Always grant the minimum permissions actually required (the **principle of least privilege**, covered again in Part 17).

0.8.7 8.7 SUID, SGID, Sticky Bit

These are special permission bits layered on top of the standard r/w/x model:

- **SUID (Set User ID)** — when set on an *executable*, the program runs with the *file owner's* privileges rather than the privileges of the user who launched it. Classic example: `/usr/bin/passwd` is owned by root and has SUID set, so any regular user can run it to change their own password (which requires writing to the root-only `/etc/shadow`), without being granted root access generally.

```
$ chmod u+s program      # set SUID
$ ls -l program
-rwsr-xr-x ...            # 's' in the owner execute slot
```

- **SGID (Set Group ID)** — similar, but for the group. On a *directory*, SGID causes new files created inside to inherit the directory's group automatically, useful for shared team folders.

```
$ chmod g+s shared_folder/
```

- **Sticky bit** — on a directory, restricts deletion so that only the file's owner (or root) can delete or rename files inside it, even if others have write access to the directory. Classic example: `/tmp` has the sticky bit set, so multiple users can create files there without being able to delete each other's.

```
$ chmod +t shared_tmp/
$ ls -ld /tmp
drwxrwxrwt ...             # 't' at the very end
```

SECURITY NOTE

SUID binaries are a well-known privilege-escalation target — an attacker who can trick a SUID-root program into executing arbitrary code inherits root privileges. Auditing for unexpected SUID binaries (`find / -perm -4000 -type f 2>/dev/null`) is a standard security-hardening and penetration-testing step, covered further in Part 23.

0.8.8 8.8 ACLs (Access Control Lists)

Standard Unix permissions only support one owner and one group per file. **ACLs** extend this, letting you grant specific permissions to *additional* individual users or groups beyond the standard three categories.

```
$ setfacl -m u:bob:rw file.txt      # grant bob read+write, beyond normal
owner/group/other
$ getfacl file.txt                  # view all ACL entries
```

0.8.9 8.9 Exercises

1. Run `id` and interpret your own UID, GID, and group memberships.

2. Create a new user with `adduser` and add them to a custom group you create with `groupadd`.
3. Create a file, then use `chmod` symbolically to remove write access for "other."
4. Recreate the same permission change using the numeric form instead.
5. Set a file to `600` and confirm with `ls -l` that only the owner has any access.
6. Find all SUID binaries on your system with `find / -perm -4000 -type f 2>/dev/null` and identify what `/usr/bin/passwd` is used for.
7. Create a shared directory, set the SGID bit, and confirm new files created inside inherit the group.
8. Explore `/tmp`'s permissions with `ls -ld /tmp` and identify the sticky bit.
9. Use `chown` to change a file's owner (requires sudo) and confirm with `ls -l`.
10. Practice fixing a "permission denied" scenario: create a script without execute permission, try to run it, then fix it with `chmod +x`.

0.8.10 Chapter Summary

- Every file has an owner, a group, and permission bits for owner/group/other, each covering read/write/execute.
- `chmod` changes permissions (symbolically or numerically); `chown`/`chgrp` change ownership.
- Numeric permissions are additive: r=4, w=2, x=1, summed per category.
- SUID/SGID/sticky bit are special-purpose bits with real security implications — understand them before setting them.
- ACLs extend permissions beyond the basic owner/group/other model when finer control is needed.

0.9 PACKAGE MANAGEMENT

0.9.1 9.1 APT, apt-get, and dpkg (Ubuntu/Mint)

Ubuntu, Mint, and Kali are all **Debian-based**, so all three share the same package management stack:

- **`dpkg`** — the low-level tool that actually installs/removes individual `.deb` package files. It does *not* resolve dependencies or download anything from the internet.
- **`apt-get` / `apt`** — higher-level tools built on top of `dpkg` that add repository management, dependency resolution, and downloading. `apt` (introduced more recently) is the friendlier, recommended interface for interactive use; `apt-get` remains common in scripts and older documentation.

```
sudo apt update                 # refresh the list of available packages/versions
sudo apt upgrade                # install available upgrades for installed packages
sudo apt install <package>      # install a new package (and its dependencies)
sudo apt remove <package>       # remove a package, keep its config files
sudo apt purge <package>        # remove a package AND its config files
sudo apt autoremove              # remove packages that were installed as dependencies
                                   #   and are no longer needed
```

```
sudo apt search <keyword>          # search available packages
apt show <package>                  # show details about a package
sudo dpkg -i package.deb            # install a locally downloaded .deb file directly
dpkg -l                             # list all installed packages
```

✓ **TIP**

`apt update` does **not** install anything — it only refreshes APT's local knowledge of what's available. You almost always run `update` immediately before `upgrade` or `install`.

0.9.2 9.2 Kali-Specific Notes

Kali uses the exact same `apt`/`dpkg` stack, since it's Debian-based. The differences are in what's *available*: Kali's repositories include hundreds of pre-packaged security tools (`kali-linux-default`, `kali-linux-large` metapackages bundle broad tool categories), and Kali tracks a rolling/testing-adjacent branch of Debian to keep those tools current.

```
sudo apt install kali-linux-headless   # a smaller, headless-friendly Kali toolset, for
example
```

0.9.3 9.3 Universal Package Formats: snap and flatpak

Beyond APT, two cross-distribution package formats have become common:

- **snap** — developed by Canonical (Ubuntu's maker), bundles an application with all its dependencies into a single sandboxed package, so it runs consistently regardless of what's installed on the base system. Pre-installed on Ubuntu by default.

```
sudo snap install <package>
snap list
sudo snap remove <package>
```

- **flatpak** — a similar sandboxing concept from a separate community project, popular especially on Mint and other distributions that intentionally avoid pre-installing snap.

```
flatpak install flathub <app-id>
flatpak list
flatpak uninstall <app-id>
```

Both trade a small amount of disk space/startup time for consistency and (in principle) improved security isolation, since each app carries its own dependencies rather than sharing system libraries.

0.9.4 9.4 Repository Concepts

A **repository** is a server hosting a collection of packages plus an index describing what's available and their checksums. Your system's list of repositories lives in `/etc/apt/sources.list` and `/etc/apt/sources.list.d/`.

- **Official repositories** — maintained by Canonical/Debian/Offensive Security; the default, trusted source.
- **PPAs (Personal Package Archives)** — Ubuntu-specific community/vendor repositories for software not in the official repos (e.g., a specific application's latest version). Adding a PPA extends trust to that maintainer — only add PPAs you have good reason to trust.

```
sudo add-apt-repository ppa:some-maintainer/software-name
sudo apt update
```

0.9.5 9.5 Dependencies

Modern software rarely stands alone — it relies on shared libraries and other packages to function. APT automatically calculates and installs the full dependency chain a package needs, and refuses actions that would leave the system in a broken, inconsistent state (unless explicitly forced, which is discouraged).

```
$ sudo apt install nginx
The following additional packages will be installed:
  libnginx-mod-http-headers-more-filter nginx-common ...
```

0.9.6 9.6 GPG Package Verification

Every official package is cryptographically signed. APT verifies these signatures before installing anything, ensuring the package genuinely came from the trusted maintainer and hasn't been tampered with in transit.

```
$ apt-key list                # (legacy; modern systems use /etc/apt/trusted.gpg.d/)
$ gpg --list-keys
```

SECURITY NOTE

This is why adding third-party repositories (PPAs, vendor repos) requires importing their signing key — without it, APT will refuse to install anything from that source. Never blindly run commands that disable signature verification (e.g., `--allow-unauthenticated`) unless you fully understand and accept the risk; doing so removes the guarantee that what you're installing is genuinely what the maintainer published.

0.9.7 9.7 Full Workflow Example

```
sudo apt update                    # 1. refresh package lists
sudo apt search wireshark          # 2. search for a package
apt show wireshark                 # 3. inspect details before installing
sudo apt install wireshark         # 4. install
wireshark --version                # 5. confirm it installed correctly
sudo apt remove wireshark          # 6. remove later if no longer needed
sudo apt autoremove                # 7. clean up now-unneeded dependencies
```

0.9.8 9.8 Exercises

1. Run `sudo apt update` and read the summary output — how many packages can be upgraded?
2. Search for a package (e.g., `htop`) with `apt search`, then inspect it with `apt show` before installing.
3. Install `tree` (used in Part 5) if you haven't already, and confirm with `tree --version`.
4. List all installed packages with `dpkg -l` and count roughly how many are on your system.
5. Remove a package you installed for practice using `apt remove`, then run `apt autoremove`.
6. Check whether `snap` is installed on your system with `snap version`, and list any installed snaps.

0.9.9 Chapter Summary

- Ubuntu, Mint, and Kali all share APT/dpkg — one consistent package-management skillset across all three.
- `apt update` refreshes package lists; `apt upgrade`/`install`/`remove` act on packages.
- snap and flatpak offer sandboxed, cross-distro packaging as an alternative/complement to APT.
- GPG signature verification is what keeps package installation trustworthy — never casually disable it.

0.10 PROCESSES AND SYSTEM MANAGEMENT

0.10.1 10.1 Process Concepts

A **process** is a running instance of a program. Every process has:

- A **PID** (Process ID) — a unique number identifying it.
- A **PPID** (Parent Process ID) — the process that launched it. Nearly every process on Linux is spawned by another, forming a tree rooted at PID 1 (traditionally `init`, now typically `systemd`).
- A state: **foreground** (attached to your terminal, blocking further input until it finishes) or **background** (running independently, freeing your terminal for other commands).

- A **daemon** — a background process, usually started at boot, providing an ongoing service (e.g., `sshd` for SSH, `cron` for scheduled tasks) rather than doing a one-off task.

Processes communicate with the kernel and each other partly through **signals** — small, standardized notifications like “please terminate” or “please reload your configuration.”

0.10.2 10.2 Process Commands

Command	Purpose
`ps`	Snapshot of current processes
`top`	Live, auto-refreshing process viewer
`htop`	Improved, colorized, interactive version of `top`
`pgrep`	Find PIDs matching a name pattern
`pidof`	Find the PID of a specific running program
`kill`	Send a signal to a process by PID
`killall`	Send a signal to all processes matching a name
`pkill`	Like `killall`, using pattern matching
`jobs`	List background/suspended jobs in the current shell
`bg`	Resume a suspended job in the background
`fg`	Bring a background job to the foreground
`nohup`	Run a command immune to hangup (keeps running after logout)
`nice`	Launch a process with an adjusted scheduling priority
`renice`	Change the priority of an already-running process

```
$ ps aux                    # all processes, all users, detailed
$ ps aux | grep firefox     # filter for a specific process
$ top                       # live view; press 'q' to quit
$ pgrep firefox
$ kill 4821                 # ask process 4821 to terminate gracefully
$ killall firefox           # ask all firefox processes to terminate
$ nice -n 10 long_task.sh   # run with lower priority (nicer to other processes)
$ renice -n 5 -p 4821       # adjust priority of an already-running process
```

Backgrounding a job:

```
$ long_running_script.sh &   # start in the background immediately (note the &)
$ jobs
```

```
[1]+  Running    long_running_script.sh &
$ fg %1                      # bring it back to the foreground
```

0.10.3 10.3 SIGTERM vs. SIGKILL

`kill` doesn't inherently mean "force quit" — by default it sends **SIGTERM (signal 15)**, a polite request asking the process to shut down cleanly (saving open files, releasing resources). A well-behaved program catches this signal and exits gracefully.

SIGKILL (signal 9) is a different matter entirely — it's handled directly by the kernel and **cannot be caught, blocked, or ignored** by the target process. The process is terminated immediately, with no chance to clean up.

```
$ kill 4821             # SIGTERM — polite request
$ kill -9 4821          # SIGKILL — immediate, forced termination
$ kill -SIGTERM 4821    # equivalent, named form
```

> ▲ **WARNING**
>
> `kill -9` should be a last resort, not a first instinct. Because the target process gets no chance to clean up, it can leave temporary files, database transactions, or locks in an inconsistent state. Always try a plain `kill` (SIGTERM) first and only escalate to `-9` if the process genuinely won't respond.

0.10.4 10.4 systemd — Modern Init and Service Management

systemd is the first process to start on modern Linux systems (PID 1) and is responsible for bringing the rest of the system up: mounting filesystems, starting network services, launching background daemons, and managing dependencies between them (e.g., "don't start the web server until networking is up").

`systemctl` — the primary tool for managing systemd services (called "units").

```
sudo systemctl start nginx        # start a service now
sudo systemctl stop nginx         # stop a service now
sudo systemctl restart nginx      # stop then start
sudo systemctl reload nginx       # reload config without a full restart (if supported)
sudo systemctl enable nginx       # start automatically on every future boot
sudo systemctl disable nginx      # stop starting automatically on boot
systemctl status nginx            # check current state, recent log lines
systemctl list-units --type=service   # list all active service units
```

`journalctl` — view logs collected by systemd's journal (covered fully in Part 16).

```
journalctl -u nginx               # logs for a specific service/unit
journalctl -u nginx -f            # follow live, like tail -f
journalctl --since "1 hour ago"
journalctl -p err                 # only error-level and above
```

Boot targets — systemd's equivalent to old-style "runlevels," defining what set of services should be active for a given system state:

Target	Roughly equivalent to	Purpose
`poweroff.target`	runlevel 0	Shut the system down
`rescue.target`	runlevel 1	Single-user, minimal recovery mode
`multi-user.target`	runlevel 3	Full multi-user system, no GUI
`graphical.target`	runlevel 5	Full system with a graphical login
`reboot.target`	runlevel 6	Restart the system

```
systemctl get-default             # what target boots by default
sudo systemctl isolate multi-user.target   # switch to a text-only session (careful — closes your desktop)
```

0.10.5 10.5 Exercises

1. Run `ps aux` and identify the process with the highest CPU usage.
2. Open `htop` (install with `sudo apt install htop` if needed) and explore its interactive interface — sort by memory usage.
3. Start a harmless long-running command in the background (`sleep 300 &`), check it with `jobs`, then bring it to the foreground with `fg`.
4. Use `pgrep` to find the PID of your shell (`bash`).
5. Practice a graceful shutdown: start `sleep 100`, find its PID with `pgrep sleep`, and terminate it with a plain `kill` (not `-9`).
6. Check the status of the SSH service with `systemctl status ssh` (or `sshd`, depending on distro).
7. View the last 20 log lines for a running service using `journalctl -u <service> -n 20`.
8. Identify your system's current boot target with `systemctl get-default`.

0.10.6 Chapter Summary

- Every running program is a process with a PID, a parent, and a state (foreground/background).
- `ps`/`top`/`htop` inspect processes; `kill`/`killall`/`pkill` send them signals.
- SIGTERM (default) requests graceful shutdown; SIGKILL (`-9`) forces immediate termination and should be a last resort.
- systemd (PID 1) manages services via `systemctl`, and its journal is inspected with `journalctl`.

0.11 STORAGE

0.11.1 11.1 Storage Inspection Commands

Command	Purpose
`lsblk`	List block devices (disks/partitions) in a tree view
`blkid`	Show filesystem type and UUID of each block device
`df`	Show space usage of mounted filesystems (Part 5)
`du`	Show space usage of specific files/directories (Part 5)
`mount`	Attach a filesystem to a directory (a “mount point”)
`umount`	Detach a mounted filesystem
`fdisk`	View/edit MBR/legacy partition tables
`parted`	View/edit partition tables (supports GPT, more modern)

```
$ lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda      8:0    0   50G  0 disk
├─sda1   8:1    0  512M  0 part /boot/efi
└─sda2   8:2    0 49.5G  0 part /

$ blkid
/dev/sda2: UUID="a1b2c3d4-..." TYPE="ext4"
```

0.11.2 11.2 Partitions and Mount Points

A **mount point** is simply a directory that a filesystem (a partition, USB drive, or network share) is attached to, making its contents accessible as part of the unified `/` tree. Before Linux can read/write a device, it must be mounted somewhere.

```
$ sudo mkdir -p /mnt/usbdrive
$ sudo mount /dev/sdb1 /mnt/usbdrive
$ ls /mnt/usbdrive
$ sudo umount /mnt/usbdrive
```

> ▲ **WARNING**
> Always `umount` a removable drive before physically disconnecting it. Unmounting flushes any data still buffered in memory to disk — pulling a USB drive while it’s still mounted risks corrupting the data on it.

0.11.3 11.3 /etc/fstab

`/etc/fstab` defines filesystems that should be mounted **automatically** at boot, rather than manually each time.

```
# <device>                              <mount point>  <type>  <options>       <dump> <pass>
UUID=a1b2c3d4-...                       /              ext4    errors=remount-ro 0      1
UUID=e5f6g7h8-...                       /boot/efi       vfat    umask=0077        0      1
```

- **device** — often referenced by UUID (from `blkid`) rather than `/dev/sdaX`, since device names can shift between boots but UUIDs don’t.

- **mount point** — where it attaches in the tree.
- **type** — filesystem type (ext4, vfat, ntfs, etc.).
- **options** — mount behavior (read-only, permissions, etc.).
- **dump/pass** — legacy backup and filesystem-check-order flags.

> ▲ **WARNING**
> A typo in `/etc/fstab` can prevent your system from booting normally (it may drop into an emergency/recovery shell). Always back up the file (`sudo cp /etc/fstab /etc/fstab.bak`) before editing, and test with `sudo mount -a` (which re-reads fstab and mounts everything listed) before rebooting.

0.11.4 11.4 Handling USB and External Disks Safely

1. Plug in the device — most desktop environments auto-mount it under `/media/<username>/<label>`
2. Confirm it with `lsblk` or `df -h`.
3. Copy files as needed.
4. **Always** unmount cleanly before removal — either via the file manager's "Eject" button, or `sudo umount /media/<username>/<label>`.

0.11.5 11.5 Destructive Command Warning

> ▲ **WARNING**
> `fdisk`, `parted`, and especially `mkfs` (make filesystem — creates a *new*, empty filesystem, destroying whatever was there before) can **permanently destroy all data** on a disk if pointed at the wrong device. Always confirm the exact device name with `lsblk` first, and never run these commands against a disk unless you are certain of its identity. Full destructive-command coverage, including safe practice guidance, is in Part 30.

```
$ sudo fdisk -l                # list partition tables (safe, read-only)
$ sudo fdisk /dev/sdb          # interactive partition editor (DESTRUCTIVE if you write
changes)
$ sudo mkfs.ext4 /dev/sdb1     # format a partition as ext4 (DESTROYS existing data on
it)
```

0.11.6 11.6 Exercises

1. Run `lsblk` and identify every disk and partition on your system.
2. Run `blkid` and note the filesystem type and UUID of your root partition.
3. Insert a USB drive (or attach a second virtual disk in your VM), mount it manually to `/mnt/test`, copy a file to it, then safely unmount it.
4. Back up your `/etc/fstab` before viewing it with `cat /etc/fstab` and identifying each field.
5. (VM only, since this is safe to practice there) Attach a second virtual disk to your VM, and use `fdisk -l` to confirm you can identify it correctly by size before doing anything further.

0.11.7 Chapter Summary

- `lsblk`/`blkid` inspect disks and partitions; `mount`/`umount` attach/detach them from the directory tree.
- `/etc/fstab` defines what mounts automatically at boot — back it up before editing.
- `fdisk`, `parted`, and `mkfs` are powerful and genuinely destructive — always verify the target device first.

0.12 NETWORKING

0.12.1 12.1 Foundations

- **LAN (Local Area Network)** — a network confined to one physical location (a home, office, or campus).
- **WAN (Wide Area Network)** — a network spanning larger distances, connecting multiple LANs (the Internet is the largest WAN).
- **NIC (Network Interface Card)** — the hardware (physical or virtual) that connects a device to a network.
- **MAC address** — a unique hardware identifier burned into (or assigned to) a NIC, used for communication at the local network (Layer 2) level, e.g. `00:1A:2B:3C:4D:5E`.

0.12.2 12.2 IP Addressing

- **IPv4** — the most common addressing scheme, written as four numbers 0–255 separated by dots, e.g. `192.168.1.10`. Roughly 4.3 billion possible addresses — a number the internet has already outgrown, driving…
- **IPv6** — a much larger address space (128-bit vs. IPv4's 32-bit), written in hexadecimal groups, e.g. `2001:0db8:85a3::8a2e:0370:7334`.
- **Private IP** — addresses reserved for internal networks, not routable on the public internet: `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`.
- **Public IP** — a globally unique, internet-routable address assigned by an ISP or cloud provider.

0.12.3 12.3 Subnetting, Subnet Mask, CIDR

A **subnet** divides a larger network into smaller, logically separated segments. A **subnet mask** (e.g., `255.255.255.0`) determines which portion of an IP address identifies the network vs. the specific host on it.

CIDR notation expresses this more compactly: `192.168.1.0/24` means the first 24 bits are the network portion, leaving 8 bits (256 addresses, 254 usable) for hosts.

CIDR	Subnet mask	Usable hosts
/24	255.255.255.0	254
/16	255.255.0.0	65,534
/8	255.0.0.0	~16.7 million

0.12.4 12.4 Gateway, Router, Switch

- **Switch** — connects devices *within* the same local network, forwarding traffic based on MAC addresses.
- **Router** — connects *different* networks together (e.g., your home LAN to your ISP's WAN), forwarding traffic based on IP addresses.
- **Gateway** — the specific address (usually your router) that a device sends traffic to when the destination is outside its local subnet.

0.12.5 12.5 DNS, DHCP, ARP, NAT

- **DNS (Domain Name System)** — translates human-friendly names (`google.com`) into IP addresses computers actually use to connect.
- **DHCP (Dynamic Host Configuration Protocol)** — automatically assigns an IP address, subnet mask, gateway, and DNS servers to a device joining a network, rather than requiring manual configuration.
- **ARP (Address Resolution Protocol)** — resolves an IP address to a MAC address *within* the local network segment ("who has 192.168.1.5? Tell 192.168.1.1").
- **NAT (Network Address Translation)** — lets many devices on a private network share a single public IP address, translating between private and public addresses as traffic crosses the boundary (this is how your entire home network shares one ISP-assigned IP).

```
Your Laptop (192.168.1.10) --> Router (NAT) --> ISP --> Internet
                                      |
                            Public IP: 203.0.113.5
```

0.12.6 12.6 Ports, TCP, UDP, ICMP

A **port** is a number (0–65535) identifying a specific service or conversation on a device — an IP address gets you to the right machine; a port gets you to the right application on it. Well-known ports: 22 (SSH), 80 (HTTP), 443 (HTTPS), 53 (DNS).

- **TCP (Transmission Control Protocol)** — connection-oriented, reliable: guarantees delivery and correct ordering, with acknowledgments and retransmission. Used where correctness matters more than raw speed (web browsing, file transfer, SSH).
- **UDP (User Datagram Protocol)** — connectionless, no delivery guarantee, no ordering guarantee — but much lower overhead. Used where speed matters more than occasional loss (video streaming, DNS queries, online gaming).
- **ICMP (Internet Control Message Protocol)** — used for network diagnostics and error reporting rather than carrying application data — this is what `ping` uses.

0.12.7 12.7 HTTP/HTTPS, SSH, FTP/SFTP (Overview)

- **HTTP** — the protocol web browsers and servers use to exchange pages/data; unencrypted by default.
- **HTTPS** — HTTP wrapped in TLS encryption, protecting data in transit; the modern web standard.

- **SSH** — encrypted remote shell access to another machine (full coverage in Part 13).
- **FTP/SFTP** — file transfer protocols; plain FTP is unencrypted and largely deprecated for anything sensitive, while SFTP (which actually runs over SSH, despite the similar name) is encrypted and the modern standard.

0.12.8 12.8 Networking Commands

```
ip addr                     # show all network interfaces and their IP addresses
ip addr show eth0           # show a specific interface
ip link                     # show interfaces (link/hardware layer)
ip link set eth0 up         # bring an interface up
ip route                    # show the routing table
ping google.com             # test reachability using ICMP
ss -tulpn                   # show listening ports and associated processes
traceroute google.com       # show the path (hop by hop) packets take to a destination
tracepath google.com        # similar to traceroute, no root required
dig google.com              # detailed DNS lookup
nslookup google.com         # simpler DNS lookup (older tool, still widely used)
host google.com             # minimal DNS lookup
curl https://example.com    # fetch a URL's content from the command line
wget https://example.com/file.zip   # download a file from the command line
arp -a                      # show the local ARP cache (IP-to-MAC mappings)
ip neigh                    # modern equivalent of arp -a
hostname                    # show this machine's hostname
nmcli device status         # show network device status (NetworkManager)
nmcli device wifi connect "SSID" password "wifipassword"   # connect to Wi-Fi from CLI
```

> ✓ **TIP**
>
> `ss` is the modern replacement for the older `netstat` command — use `ss -tulpn` to see every listening TCP/UDP port and which process owns it, a command you'll use constantly for both troubleshooting and security auditing.

0.12.9 12.9 Deep Dive: What Happens When You Run `ping google.com`

This single command touches almost every concept in this chapter. Here's the full journey:

```
1. DNS RESOLUTION
   Your system checks: is "google.com" already cached? If not, it
   queries a DNS server (often your router, or 8.8.8.8/1.1.1.1)
   to resolve "google.com" into an IP address, e.g. 142.250.premium.
                    |
                    v
2. ROUTING DECISION
   Your kernel checks its routing table (`ip route`): is the
   destination IP on my local subnet, or does it need to go
   through my default gateway?
```

```
   -> Since google.com is not local, it's routed via the gateway.
                    |
                    v
3. ARP / NEIGHBOR DISCOVERY
   To actually send a packet to the gateway, your machine needs
   the gateway's MAC address on the local network. It checks its
   ARP cache, or broadcasts an ARP request if not cached.
                    |
                    v
4. PACKET CONSTRUCTION
   An ICMP "Echo Request" packet is built, wrapped in an IP
   packet addressed to google.com's IP, wrapped in an Ethernet
   frame addressed to the gateway's MAC address.
                    |
                    v
5. TRANSMISSION THROUGH ROUTERS
   The gateway receives the frame, strips the Ethernet layer,
   looks at the IP destination, and forwards it toward the next
   hop -- repeating across however many routers sit between you
   and Google's infrastructure.
                    |
                    v
6. DESTINATION RESPONDS
   Google's server receives the Echo Request and replies with an
   ICMP "Echo Reply," which retraces a path back to you (not
   necessarily the identical route).
                    |
                    v
7. YOUR TERMINAL DISPLAYS THE RESULT
   $ ping google.com
   64 bytes from 142.250.x.x: icmp_seq=1 ttl=115 time=14.2 ms
```

Every field in that output tells you something: `icmp_seq` (packet sequence number, useful for spotting loss), `ttl` (Time To Live — how many more router hops the packet could survive, a rough distance indicator), and `time` (round-trip latency).

0.12.10 12.10 Exercises

1. Run `ip addr` and identify your machine's local IP address and subnet.
2. Run `ip route` and identify your default gateway.
3. Run `ping -c 4 google.com` (the `-c 4` limits it to 4 packets) and interpret the output.
4. Run `dig google.com` and identify the resolved IP address(es).
5. Use `ss -tulpn` to list all currently listening ports on your machine and identify what's using port 22 (if SSH is installed).
6. Run `traceroute google.com` (install with `sudo apt install traceroute` if needed) and count the number of hops.
7. Use `curl -I https://example.com` to fetch just the HTTP headers of a website (the `-I` flag requests headers only).

8. Explain, in your own words, the difference between TCP and UDP, and give one real-world example of each.
9. Check your ARP cache with `ip neigh` after pinging a device on your local network.
10. Use `nmcli device status` to see your network interfaces' current connection state.

0.12.11 Chapter Summary

- IP addresses identify machines; ports identify specific services on them; MAC addresses identify hardware on the local segment.
- DNS translates names to IPs; DHCP automates IP assignment; ARP resolves IPs to MACs locally; NAT lets many private devices share one public IP.
- TCP is reliable and ordered; UDP is fast but makes no delivery guarantees.
- `ping google.com` exercises DNS, routing, ARP, and ICMP in a single command — understanding that flow underpins nearly everything else in this chapter.

0.13 SSH — SECURE SHELL

0.13.1 13.1 SSH Commands

Command	Purpose
`ssh`	Open a secure remote shell session on another machine
`scp`	Securely copy files between machines over SSH
`sftp`	Interactive secure file transfer over SSH
`ssh-keygen`	Generate a public/private SSH key pair

```
$ ssh student@192.168.1.20              # connect to a remote machine
$ ssh -p 2222 student@192.168.1.20       # connect on a non-default port
$ scp report.pdf student@192.168.1.20:/home/student/     # copy a file TO a remote
machine
$ scp student@192.168.1.20:/home/student/report.pdf .    # copy a file FROM a remote
machine
$ sftp student@192.168.1.20              # open an interactive file-transfer session
```

0.13.2 13.2 Client vs. Server

- The **SSH server** (`sshd`, the "SSH daemon") runs on the machine you want to connect *to*, listening on port 22 by default.
- The **SSH client** (`ssh`) runs on the machine you're connecting *from*.

```
sudo apt install openssh-server      # install the server on the machine to be accessed
sudo systemctl status ssh            # confirm it's running
sudo systemctl enable ssh            # start it automatically on boot
```

0.13.3 13.3 Password vs. Key Authentication

- **Password authentication** — you type your account password on each connection. Simple, but vulnerable to brute-force guessing if exposed to the internet without protection.
- **Key authentication** — a cryptographic key pair (public + private) replaces the password entirely. This is the modern, recommended standard: far stronger, and immune to password-guessing attacks.

0.13.4 13.4 Generating and Using SSH Keys

```
$ ssh-keygen -t ed25519 -C "your_email@example.com"
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/student/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
```

This creates two files: - **`~/.ssh/id_ed25519`** — your **private key**. Never share this. Ever. - **`~/.ssh/id_ed25519.pub`** — your **public key**. Safe to share; this is what you give to servers you want to access.

Copying your public key to a server:

```
$ ssh-copy-id student@192.168.1.20
```

This appends your public key to `~/.ssh/authorized_keys` on the remote machine — any connection presenting the matching private key is now trusted without a password.

Correct permissions matter:

```
$ chmod 700 ~/.ssh
$ chmod 600 ~/.ssh/id_ed25519
$ chmod 644 ~/.ssh/id_ed25519.pub
$ chmod 600 ~/.ssh/authorized_keys
```

❧ **SECURITY NOTE**

OpenSSH will actively **refuse to use a private key** with overly permissive file permissions (e.g., world-readable), precisely because a private key readable by other users on the system defeats its entire purpose. `600` (owner read/write only) is the required permission for private keys and `authorized_keys`.

0.13.5 13.5 Basic SSH Configuration

Server-side (`/etc/ssh/sshd_config`) — key hardening directives (edit as root, then `sudo systemctl restart ssh`):

```
PermitRootLogin no          # never allow direct root login over SSH
PasswordAuthentication no   # once keys are set up, disable password login entirely
Port 2222                    # optional: run on a non-standard port
```

Client-side (`~/.ssh/config`) — define shortcuts for frequently used connections:

```
Host myserver
    HostName 192.168.1.20
    User student
    Port 22
    IdentityFile ~/.ssh/id_ed25519
```

```
$ ssh myserver     # now equivalent to ssh -p 22 student@192.168.1.20
```

0.13.6 13.6 Secure SSH Practices

SECURITY NOTE

- **Disable root login** (`PermitRootLogin no`) — force administrators to log in as a regular user and `sudo` when needed, preserving an audit trail.
- **Prefer key-only authentication** once set up — disable `PasswordAuthentication` entirely to eliminate brute-force password guessing as a viable attack.
- **Keep `authorized_keys` clean** — remove keys for people/devices that no longer need access.
- **Consider fail2ban** (previewed here, covered fully in Part 17) — automatically bans IP addresses after repeated failed login attempts.
- **Change the default port** as a mild deterrent against opportunistic mass-scanning bots (not a substitute for the measures above, but a reasonable extra layer).

0.13.7 13.7 Lab: Local VM-to-VM SSH Setup

★ LAB

Objective: Set up passwordless, key-based SSH access between two local virtual machines.
Requirements: Two VMs on the same host-only or NAT network (e.g., an Ubuntu server VM and your main Ubuntu/Mint VM).
Steps: 1. On the target VM, install and enable the SSH server: `sudo apt install openssh-server && sudo systemctl enable --now ssh`. 2. Find its IP address with `ip addr`. 3. On your client VM, generate a key pair: `ssh-keygen -t ed25519`. 4. Copy your public key across: `ssh-copy-id student@<target-ip>`. 5. Connect: `ssh student@<target-ip>` — it should no longer prompt for a password. 6. Copy a test file across with `scp testfile.txt student@<target-ip>:~/`. 7. On the target VM, edit `/etc/ssh/sshd_config` to set `PasswordAuthentication no`, then `sudo systemctl restart ssh`. 8. Confirm password login now fails, while your key-based login still works.
Expected outcome: You can SSH into the target VM without typing a password, and password-based login attempts are rejected.
Common mistakes: Forgetting to restart `sshd` after editing its config; incorrect permissions on `~/.ssh` or the private key (SSH will silently refuse to use it); typos in

the target IP address.

0.13.8 13.8 Exercises

1. Generate an SSH key pair with `ssh-keygen -t ed25519`.
2. Inspect the permissions on your `~/.ssh` directory and files with `ls -la ~/.ssh`.
3. Set up a second local VM and connect to it over SSH using a password first, then switch to key-based auth.
4. Create a `~/.ssh/config` entry for that VM and connect using just its shortcut name.
5. Use `scp` to copy a file to the remote VM, then `sftp` to browse its filesystem interactively.

0.13.9 Chapter Summary

- SSH provides encrypted remote shell access; `scp`/`sftp` provide encrypted file transfer over the same mechanism.
- Key-based authentication is stronger than passwords and is the modern standard — generate a key pair with `ssh-keygen`, distribute the public key with `ssh-copy-id`.
- Private keys and `authorized_keys` require strict `600` permissions, enforced by OpenSSH itself.
- Hardening SSH (disabling root login and password auth, considering fail2ban) is one of the highest-value security steps you can take on any internet-facing Linux machine.

0.14 THE BASH SHELL

0.14.1 14.1 What Is Bash?

Bash (Bourne Again SHell) is the default command interpreter on Ubuntu, Mint, and Kali. Everything you've typed so far in this book has been interpreted by bash. Beyond running individual commands interactively, bash is also a full scripting language — Part 15 builds on everything in this chapter to write complete automation scripts.

0.14.2 14.2 Variables, Environment Variables, PATH, Aliases

Shell variables exist only within your current shell session:

```
$ name="student"
$ echo $name
student
```

Environment variables are shell variables marked to be passed on to any program the shell launches — by convention written in uppercase:

```
$ export GREETING="hello"
$ printenv GREETING
hello
```

PATH is the most important environment variable for daily use — a colon-separated list of directories the shell searches, in order, when you type a command name:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

This is exactly what makes `ls` work without typing `/usr/bin/ls` — the shell searches each PATH directory until it finds a matching executable.

Aliases create shortcuts for longer commands:

```
$ alias ll='ls -la'
$ ll                    # now runs `ls -la`
```

Add aliases to `~/.bashrc` to make them permanent across sessions.

0.14.3 14.3 Pipes, Redirects, and Standard Streams

Every process has three standard I/O streams:

Stream	Number	Default destination
stdin (standard input)	0	Keyboard
stdout (standard output)	1	Terminal screen
stderr (standard error)	2	Terminal screen

Redirects change where these streams go:

```
$ ls > filelist.txt          # send stdout to a file (overwrite)
$ ls >> filelist.txt         # append stdout to a file
$ sort < filelist.txt        # take stdin from a file
$ command 2> errors.log      # send stderr to a file
$ command > out.log 2>&1     # send BOTH stdout and stderr to the same file
$ command > out.log 2>errors.log   # send them to separate files
```

Pipes (|) connect the stdout of one command directly to the stdin of the next, letting you chain small tools into powerful combinations:

```
$ ps aux | grep firefox | wc -l
```

This runs three commands in sequence: `ps aux` lists processes, `grep firefox` filters for lines mentioning firefox, and `wc -l` counts the resulting lines — three simple tools composed into one useful answer ("how many firefox processes are running?").

✓ **TIP**

This philosophy — small, focused tools connected with pipes rather than one giant do-everything program — is one of the defining ideas of Unix, and it's why the shell remains such a powerful environment decades later.

0.14.4 14.4 Text-Processing Commands

Command	Purpose	Example
`echo`	Print text	`echo "Hello"`
`printf`	Print formatted text (more control than echo)	`printf "%s is %d\n" "age" 30`
`env`	Show all environment variables	`env`
`export`	Mark a variable as an environment variable	`export NAME=value`
`grep`	Search text for a pattern	`grep "error" log.txt`
`cut`	Extract columns/fields from lines	`cut -d',' -f1 data.csv`
`sort`	Sort lines	`sort names.txt`
`uniq`	Remove/count adjacent duplicate lines	`sort file.txt | uniq -c`
`wc`	Count lines/words/characters	`wc -l file.txt`
`tr`	Translate/replace characters	`tr 'a-z' 'A-Z'`
`xargs`	Build/execute commands from stdin	`find . -name "*.tmp" | xargs rm`
`tee`	Write stdout to a file *and* pass it through	`command | tee output.log`
`sed`	Stream editor — find/replace, and more	`sed 's/foo/bar/g' file.txt`
`awk`	Pattern-scanning and text-processing language	`awk '{print $1}' file.txt`

Worked examples:

```
# Count how many lines in a log file contain "ERROR"
$ grep -c "ERROR" app.log

# Show only unique IP addresses from an access log, sorted
$ awk '{print $1}' access.log | sort | uniq

# Replace all occurrences of "http" with "https" in a file
$ sed 's/http:/https:/g' urls.txt
```

```
# Extract the second column of a CSV file
$ cut -d',' -f2 data.csv

# Count how many processes each user is running
$ ps aux | awk '{print $1}' | sort | uniq -c | sort -rn

# Save command output to a file while still viewing it live
$ ping -c 5 google.com | tee ping_results.txt

# Delete every .tmp file found under the current directory
$ find . -name "*.tmp" | xargs rm -v
```

✓ **TIP**

`grep`, `sed`, and `awk` are three of the most powerful tools in this book. `grep` finds lines; `sed` transforms text (especially find-and-replace); `awk` is a small but complete language for processing structured, column-based text. Mastering these three alone will make you dramatically faster at log analysis, data cleanup, and scripting.

0.14.5 14.5 Practical Pipelines

```
# Find the 5 largest files under the current directory
$ du -ah . | sort -rh | head -5

# Show the top 5 most-frequently-occurring words in a text file
$ tr -s ' ' '\n' < document.txt | sort | uniq -c | sort -rn | head -5

# List all currently logged-in usernames, deduplicated
$ who | awk '{print $1}' | sort -u

# Show disk usage percentage for each mounted filesystem, sorted descending
$ df -h | tail -n +2 | sort -k5 -rn
```

0.14.6 14.6 Exercises

1. Set an environment variable and confirm a subprocess (like `bash -c 'echo $MYVAR'`) can see it.
2. Redirect the output of `ls -la` to a file, then view it with `cat`.
3. Use `grep` to search for a keyword in `/var/log/dpkg.log` (or any log file present on your system).
4. Chain `ps aux`, `grep`, and `wc -l` to count how many processes a specific user is running.
5. Use `sed` to replace a word in a text file you create.
6. Use `awk` to print just the first column of `/etc/passwd` (usernames).
7. Use `sort` and `uniq -c` together to count occurrences of unique lines in a file.
8. Use `tee` to save command output to a file while still seeing it on screen.
9. Add a personal alias to `~/.bashrc`, then reload it with `source ~/.bashrc` and test it.
10. Use `xargs` to delete a set of test files matched by `find`.

0.14.7 Chapter Summary

- Bash variables, `exported` environment variables, and `PATH` control how commands are found and how data is shared with subprocesses.
- Redirects (`>`, `>>`, `<`, `2>`, `2>&1`) control where stdin/stdout/stderr go; pipes (`|`) chain commands together.
- `grep`, `sed`, `awk`, `cut`, `sort`, `uniq`, `wc`, `tr`, `xargs`, and `tee` form the essential Unix text-processing toolkit — small tools combined into powerful pipelines.

0.15 BASH SCRIPTING

0.15.1 15.1 Getting Started

A bash script is just a text file containing a sequence of commands, executed in order.

```
#!/bin/bash
echo "Hello, Linux!"
```

The first line, `#!/bin/bash`, is called a **shebang** — it tells the kernel which interpreter should run this file when executed directly. Without it, the file would only work if you explicitly ran `bash script.sh`.

```
$ chmod +x hello.sh     # make it executable
$ ./hello.sh            # run it (the ./ is required unless it's in your PATH)
Hello, Linux!
```

0.15.2 15.2 Variables, Input, and Arguments

```
#!/bin/bash
name="Linux"
echo "Hello, $name!"

echo "What's your name?"
read user_name
echo "Nice to meet you, $user_name!"
```

Command-line arguments are automatically available as special variables:

Variable	Meaning
`$1`, `$2`, …	First, second, … positional argument
`$@`	All arguments, as separate words
`$#`	Number of arguments passed
`$0`	The script's own name

```
#!/bin/bash
echo "Script name: $0"
echo "First argument: $1"
echo "Total arguments: $#"
echo "All arguments: $@"
```

```
$ ./args.sh alpha beta gamma
Script name: ./args.sh
First argument: alpha
Total arguments: 3
All arguments: alpha beta gamma
```

0.15.3 15.3 Exit Codes

Every command, including your own scripts, returns a numeric **exit code** when it finishes: `0` means success, anything else (1–255) signals some kind of failure, with the specific number often indicating the type of error.

```
$ ls /nonexistent
$ echo $?
2
```

`$?` always holds the exit code of the most recently completed command — check it immediately, before running anything else.

0.15.4 15.4 Conditionals

```
#!/bin/bash
count=5

if [ "$count" -gt 10 ]; then
    echo "Count is large"
elif [ "$count" -gt 0 ]; then
    echo "Count is positive but small"
else
    echo "Count is zero or negative"
fi
```

Common test operators:

Operator	Meaning
`-eq -ne`	equal / not equal (numbers)
`-gt -lt -ge -le`	greater/less than (or equal), numbers
`= !=`	equal / not equal (strings)
`-z`	string is empty
`-n`	string is non-empty

Operator	Meaning
`-f`	file exists and is a regular file
`-d`	directory exists
`-x`	file exists and is executable

0.15.5 15.5 Loops

```
# for loop over a list
for fruit in apple banana cherry; do
    echo "I like $fruit"
done

# for loop over a range
for i in {1..5}; do
    echo "Number: $i"
done

# for loop over files
for file in *.txt; do
    echo "Found: $file"
done

# while loop
count=1
while [ "$count" -le 5 ]; do
    echo "Count: $count"
    count=$((count + 1))
done
```

0.15.6 15.6 Functions and case

```
greet() {
    local name=$1
    echo "Hello, $name!"
}
greet "World"

check_status() {
    case $1 in
        start) echo "Starting service..." ;;
        stop)  echo "Stopping service..." ;;
        *)     echo "Unknown command: $1" ;;
    esac
}
check_status "start"
```

0.15.7 15.7 Arrays, Strings, and File Tests

```
# Arrays
fruits=("apple" "banana" "cherry")
echo "${fruits[0]}"        # apple
echo "${fruits[@]}"        # all elements
echo "${#fruits[@]}"       # array length (3)

# String manipulation
str="Hello, Linux!"
echo "${#str}"              # length: 13
echo "${str,,}"              # lowercase
echo "${str^^}"              # uppercase
echo "${str/Linux/World}"    # replace: Hello, World!

# File test operators, used in conditionals
if [ -f "/etc/passwd" ]; then
    echo "File exists"
fi
if [ -d "/home" ]; then
    echo "Directory exists"
fi
```

0.15.8 15.8 Error Handling

```
#!/bin/bash
set -e                 # exit immediately if any command fails
set -u                 # error on use of an undefined variable
set -o pipefail        # a pipeline fails if ANY command in it fails, not just the last

trap 'echo "An error occurred on line $LINENO"; exit 1' ERR
```

> ✓ **TIP**
> Starting production-quality scripts with `set -euo pipefail` is a widely-recommended habit — it turns silent failures (a common source of scripts that "seem to work" but quietly do the wrong thing) into loud, immediate ones.

0.15.9 15.9 Seven Full Projects

Project 1 — System Information Script

```
#!/bin/bash
# sysinfo.sh - Print a summary of key system information

echo "===== System Information ====="
echo "Hostname:     $(hostname)"
echo "Kernel:       $(uname -r)"
```

```
echo "Uptime:       $(uptime -p)"
echo "CPU:          $(nproc) cores"
echo "Memory:       $(free -h | awk '/^Mem:/ {print $3 " / " $2}')"
echo "Disk (root):  $(df -h / | awk 'NR==2 {print $3 " / " $2 " (" $5 " used)"}')"
echo "IP Address:   $(hostname -I | awk '{print $1}')"
echo "================================"
```

Line-by-line: `$(...)` is **command substitution** — it runs the command inside and drops its output directly into the string. `uptime -p` prints uptime in "pretty" form (e.g., "up 2 hours"). `free -h` shows memory in human-readable units; the `awk` call pulls used/total memory from the "Mem:" line. `df -h /` reports root disk usage; `NR==2` tells awk to only process the second line (the data row, skipping the header).

Project 2 — Disk Usage Monitor

```
#!/bin/bash
# disk_monitor.sh - Alert if any filesystem exceeds a usage threshold
THRESHOLD=80

df -h --output=target,pcent | tail -n +2 | while read -r mount pcent; do
    usage=${pcent%\%}          # strip the trailing % sign
    if [ "$usage" -ge "$THRESHOLD" ]; then
        echo "WARNING: $mount is at ${usage}% usage (threshold: ${THRESHOLD}%)"
    fi
done
```

Line-by-line: `df -h --output=target,pcent` limits `df`'s columns to just the mount point and usage percentage. `tail -n +2` skips the header row. The `while read` loop processes each line; `${pcent%\%}` strips the trailing `%` character using bash's parameter expansion, leaving a plain number for comparison.

Project 3 — Backup Script

```
#!/bin/bash
# backup.sh - Archive a directory with a timestamped filename
set -euo pipefail

SOURCE_DIR="$1"
BACKUP_DIR="$HOME/backups"
TIMESTAMP=$(date +"%Y-%m-%d_%H-%M-%S")
BACKUP_FILE="$BACKUP_DIR/backup_${TIMESTAMP}.tar.gz"

if [ -z "$SOURCE_DIR" ]; then
    echo "Usage: $0 <directory-to-back-up>"
    exit 1
fi

mkdir -p "$BACKUP_DIR"
```

```
tar -czf "$BACKUP_FILE" "$SOURCE_DIR"
echo "Backup created: $BACKUP_FILE ($(du -h "$BACKUP_FILE" | cut -f1))"
```

Line-by-line: `$1` is required as the directory to back up; if empty (`-z`), the script prints usage and exits with a non-zero code. `date +"%Y-%m-%d_%H-%M-%S"` builds a sortable, unique timestamp for the filename. `tar -czf` compresses the source directory into a single archive.

Project 4 — Log Analyzer

```
#!/bin/bash
# log_analyzer.sh - Summarize error counts in a log file
LOG_FILE="$1"

if [ ! -f "$LOG_FILE" ]; then
    echo "Error: file '$LOG_FILE' not found."
    exit 1
fi

echo "===== Log Summary: $LOG_FILE ====="
echo "Total lines:   $(wc -l < "$LOG_FILE")"
echo "ERROR lines:   $(grep -c "ERROR" "$LOG_FILE")"
echo "WARNING lines: $(grep -c "WARNING" "$LOG_FILE")"
echo ""
echo "Top 5 most common ERROR messages:"
grep "ERROR" "$LOG_FILE" | sort | uniq -c | sort -rn | head -5
```

Line-by-line: `[ ! -f "$LOG_FILE" ]` checks the file does *not* exist (note the `!`), exiting early with a clear message if so — this kind of input validation is what separates a robust script from a fragile one. The final pipeline finds all ERROR lines, sorts them so identical messages become adjacent, `uniq -c` counts each unique message, and `sort -rn` orders by count descending.

Project 5 — Network Diagnostic Script

```
#!/bin/bash
# netcheck.sh - Basic network health check
TARGET="${1:-google.com}"

echo "===== Network Diagnostic: $TARGET ====="

echo -n "1. Local interface: "
ip -brief addr show | grep -v "^lo" | awk '{print $1, $3}' | head -1

echo -n "2. Default gateway: "
ip route | awk '/default/ {print $3}'

echo -n "3. DNS resolution: "
```

```
if host "$TARGET" > /dev/null 2>&1; then
    echo "OK ($(host "$TARGET" | awk '/has address/ {print $4; exit}'))"
else
    echo "FAILED"
fi

echo -n "4. Reachability (ping): "
if ping -c 2 -W 2 "$TARGET" > /dev/null 2>&1; then
    echo "OK"
else
    echo "FAILED"
fi
```

Line-by-line: `${1:-google.com}` uses a default value if no argument is supplied. `> /dev/null 2>&1` silences a command's output while still letting the script check its exit code with `if command; then`. `/dev/null` is a special device file that discards anything written to it — the standard way to suppress output you don't need.

Project 6 — User Management Helper

```
#!/bin/bash
# user_helper.sh - Create a user with a home directory and a specified group
set -euo pipefail

USERNAME="$1"
GROUPNAME="${2:-users}"

if [ "$EUID" -ne 0 ]; then
    echo "This script must be run with sudo."
    exit 1
fi

if id "$USERNAME" &>/dev/null; then
    echo "User '$USERNAME' already exists."
    exit 1
fi

getent group "$GROUPNAME" > /dev/null || groupadd "$GROUPNAME"
useradd -m -g "$GROUPNAME" -s /bin/bash "$USERNAME"
passwd "$USERNAME"

echo "User '$USERNAME' created and added to group '$GROUPNAME'."
```

Line-by-line: `$EUID` is the *effective* UID of whoever is running the script — checking it's `0` (root) enforces that this script must be run with `sudo`. `id "$USERNAME" &>/dev/null` checks whether the user already exists without printing anything, using its exit code. `getent group ... || groupadd ...` creates the group only if it doesn't already exist.

Project 7 — Service Health Checker

```
#!/bin/bash
# service_check.sh - Verify a list of critical services are running
SERVICES=("ssh" "cron" "networking")

echo "===== Service Health Check ====="
all_ok=true

for svc in "${SERVICES[@]}"; do
    if systemctl is-active --quiet "$svc"; then
        echo "[OK]     $svc is running"
    else
        echo "[FAILED] $svc is NOT running"
        all_ok=false
    fi
done

if [ "$all_ok" = true ]; then
    echo "All services healthy."
    exit 0
else
    echo "One or more services need attention."
    exit 1
fi
```

Line-by-line: `systemctl is-active --quiet "$svc"` checks a service's status without printing anything, relying purely on its exit code (0 = active) inside the `if`. The script tracks overall health in `all_ok` and returns a meaningful exit code at the end — useful if this script is itself called by a monitoring system or cron job.

0.15.10 15.10 Exercises

1. Modify Project 1 (System Information) to also report the currently logged-in username.
2. Modify Project 2 (Disk Usage Monitor) to accept the threshold as a command-line argument instead of hardcoding it.
3. Run Project 3 (Backup) against a test directory and confirm the resulting archive extracts correctly.
4. Modify Project 4 (Log Analyzer) to also report the top 5 most common WARNING messages.
5. Run Project 5 (Network Diagnostic) against a target that doesn't exist (e.g., `notarealsite.invali` and confirm it correctly reports failure.
6. Trace through Project 6 (User Management) and explain, in your own words, why `set -euo pipefail` matters for a script that creates system accounts.
7. Add a new service to Project 7's `SERVICES` array and re-run it.
8. Write your own script from scratch that accepts a filename as `$1` and prints whether it's a file, a directory, or doesn't exist.

0.15.11 Chapter Summary

- Scripts start with a shebang (`#!/bin/bash`) and must be made executable (`chmod +x`) before running directly.
- `$1`/`$@`/`$#`/`$?` give you arguments and exit-status information; `if`/`case`/`for`/`while` provide control flow.
- `set -euo pipefail` is a strong default for production-quality scripts — fail loudly and immediately rather than silently continuing after an error.
- The seven projects in this chapter are realistic, reusable building blocks — the kind of scripts real Linux administrators keep in their toolkits.

0.16 LOGS AND TROUBLESHOOTING

0.16.1 16.1 /var/log Tour

Nearly every subsystem on a Linux machine writes a record of what it's doing to `/var/log`. Some of the most useful:

File/directory	Contents
`/var/log/syslog` (Debian family)	General system activity log
`/var/log/auth.log`	Authentication events — logins, sudo usage, SSH attempts
`/var/log/kern.log`	Kernel messages
`/var/log/dpkg.log`	Package installation/removal history
`/var/log/apt/history.log`	APT operation history
`/var/log/Xorg.0.log`	Graphical display server log
`/var/log/nginx/`, `/var/log/apache2/`	Web server access/error logs (if installed)

Many modern services log through **systemd's journal** instead of (or in addition to) plain text files — that's what `journalctl` reads.

0.16.2 16.2 Core Log-Reading Commands

```
journalctl                      # view the entire systemd journal, oldest first
journalctl -e                   # jump to the end (most recent entries)
journalctl -f                   # follow live, like tail -f
journalctl -u ssh                # logs for a specific unit/service
journalctl --since "2026-07-28 08:00"
journalctl --since "1 hour ago"
journalctl -p err                # only priority "error" and worse
journalctl -k                    # kernel messages only (like dmesg)

dmesg                            # kernel ring buffer -- boot and hardware messages
dmesg | tail -20                 # most recent kernel messages
dmesg -T                         # human-readable timestamps

tail -f /var/log/syslog          # follow a plain-text log live
```

```
grep -i "error" /var/log/syslog  # search a log for a keyword, case-insensitive
```

> ✓ **TIP**
> Combine `journalctl -u <service> -f` with reproducing a problem in real time (e.g., restarting the service or triggering the failing action in another terminal) — watching logs live as an issue happens is one of the fastest ways to pinpoint a root cause.

0.16.3 16.3 Troubleshooting Decision Trees

Boot Problems

```
System won't boot
  |
  +-- Stuck at GRUB / no OS listed?
  |     --> Boot from live USB, run Boot-Repair, or reinstall GRUB
  |
  +-- GRUB loads, then hangs/crashes during startup?
  |     --> Boot into recovery mode (GRUB menu), check
  |         `journalctl -b` (this boot's log) for the failing service
  |
  +-- Boots to emergency/rescue shell?
        --> Usually a bad /etc/fstab entry -- check with
            `cat /etc/fstab`, comment out the suspicious line,
            `mount -a` to test, then fix and reboot
```

Network Problems

```
No network connectivity
  |
  +-- `ip addr` shows no IP address?
  |     --> Check cable/Wi-Fi; try `sudo dhclient` or
  |         `nmcli device connect <interface>`
  |
  +-- Has an IP, but `ping <gateway>` fails?
  |     --> Local network/driver issue; check `ip link` is UP,
  |         check physical connection or Wi-Fi driver
  |
  +-- Can ping gateway, but `ping 8.8.8.8` fails?
  |     --> Router/ISP issue, or firewall blocking outbound traffic
  |
  +-- Can ping 8.8.8.8, but `ping google.com` fails?
        --> DNS problem (see below)
```

DNS Problems

```
Can reach IPs but not domain names
```

```
  |
  +-- `dig google.com` returns nothing / times out?
  |     --> DNS server unreachable; check /etc/resolv.conf,
  |         try `dig google.com @8.8.8.8` to bypass your
  |         configured resolver and isolate the problem
  |
  +-- Resolves to the wrong/unexpected IP?
        --> Check /etc/hosts for a stale manual entry,
            check for DNS cache poisoning if on an untrusted network
```

Disk Full

```
"No space left on device"
  |
  +-- `df -h` confirms which filesystem is full
  |
  +-- `du -sh /* 2>/dev/null | sort -rh | head` from that
  |     mount point to find what's consuming space
  |
  +-- Common culprits: /var/log (uncontrolled log growth),
  |     old kernels (`dpkg -l | grep linux-image`),
  |     Docker images/containers, /tmp
  |
  +-- Also check inode exhaustion separately: `df -i`
        (a disk can be "full" on inodes even with free space)
```

High CPU

```
System sluggish, high CPU
  |
  +-- `top` or `htop`, sort by CPU (press 'P' in top)
  |
  +-- Identify the offending PID and process name
  |
  +-- Is it expected (a legitimate build/render job) or unexpected?
  |     --> Unexpected process names, especially from
  |         unfamiliar paths, warrant a security review (Part 17)
  |
  +-- `renice` to lower its priority, or `kill` it if safe to do so
```

High Memory Usage

```
System sluggish, swapping heavily
  |
  +-- `free -h` -- how much swap is in use?
  |
  +-- `top`/`htop` sorted by memory (press 'M' in top)
```

```
  |
  +-- Identify the top consumer; check for a memory leak
  |     (steadily climbing usage over time in the same process)
  |
  +-- Consider adding swap, or addressing the root cause
        (undersized VM, misbehaving application)
```

Permission Denied

```
"Permission denied" error
  |
  +-- `ls -l` the file/directory -- what are the actual permissions?
  |
  +-- `whoami` and `id` -- are you the owner? In the right group?
  |
  +-- Does the action need root? (`sudo` the command instead
  |     of changing permissions, if it's a one-off admin task)
  |
  +-- Fix ownership (`chown`) or permissions (`chmod`) only if
        this is genuinely meant to be a persistent access change
```

Application Crashes

```
Application won't start / crashes immediately
  |
  +-- Run it from the terminal, not a desktop icon, to see
  |     error output directly
  |
  +-- `journalctl -xe` for recent, detailed system errors
  |
  +-- Check for missing dependencies: `ldd <binary>` shows
  |     required shared libraries and flags any that are missing
  |
  +-- Check for a corrupted config file in ~/.config/ -- try
        renaming it temporarily to test with defaults
```

Service Failures

```
A systemd service won't start
  |
  +-- `systemctl status <service>` -- read the recent log lines
  |     it shows directly
  |
  +-- `journalctl -u <service> -e` for the full recent history
  |
  +-- `sudo systemctl restart <service>` and immediately re-check status
  |
```

```
  +-- Test the underlying config file syntax if the service
        has a "config test" mode (e.g., `nginx -t`)
```

Package Installation Errors

```
`apt install` fails
  |
  +-- "Unable to locate package" --> `sudo apt update` first
  |
  +-- "Unmet dependencies" --> `sudo apt --fix-broken install`
  |
  +-- "Could not get lock" --> another apt/dpkg process is running;
  |     wait for it, or check for a stuck process holding the lock
  |
  +-- GPG/signature errors --> verify the repository's signing key
        is correctly installed (Part 9); don't bypass verification
```

0.16.4 16.4 Exercises (Scenario-Based)

1. Simulate a disk-full scenario safely in a VM by creating a large dummy file (`fallocate -l 1G bigfile.img`), then use `df -h` and `du` to locate and clean it up.
2. Deliberately misconfigure a line in a *copy* of `/etc/fstab` (never the real file) and practice identifying the error without applying it.
3. Use `journalctl -u ssh` to review recent SSH authentication activity on your system.
4. Stop the SSH service (`sudo systemctl stop ssh`), confirm the failure with `systemctl status ssh`, then restart it and confirm recovery.
5. Practice the "Permission Denied" decision tree: create a script without execute permission, attempt to run it, diagnose with `ls -l`/`whoami`, and fix it.
6. Use `dmesg -T | tail -30` to review the most recent kernel messages on your system and identify any warnings.

0.16.5 Chapter Summary

- `/var/log` and `journalctl` are your two primary sources of truth when something goes wrong — learn both.
- Troubleshooting is fastest when approached as a decision tree: narrow the problem down layer by layer (is it DNS? routing? permissions? the application itself?) rather than guessing.
- `df`/`du` for disk-full issues, `top`/`htop` for CPU/memory issues, `ls -l`/`id` for permission issues, `systemctl status`/`journalctl -u` for service issues.

0.17 LINUX SECURITY

0.17.1 17.1 Foundational Principles

- **Principle of Least Privilege** — every user, process, and service should have only the minimum access required to do its job, nothing more. This single idea underlies

almost every other practice in this chapter.

- **Root security** — root can do anything, which means anything that *can* run as root (a compromised service, a careless script) becomes a full system compromise. Minimize what runs as root; prefer `sudo` for individual commands over persistent root sessions.
- **File permissions** — reviewed in depth in Part 8; misconfigured permissions (especially world-writable files, or unnecessary SUID binaries) are a common, preventable weakness.
- **Password security** — strong, unique passwords; consider `passwd`'s aging policies (`chage`) for shared/managed systems; prefer SSH keys over passwords entirely where possible (Part 13).

0.17.2 17.2 Firewalls

A firewall controls what network traffic is allowed in and out of a machine, based on rules matching ports, protocols, and addresses.

`ufw` (Uncomplicated Firewall) — a friendly front-end for the underlying netfilter/iptables framework, the default recommended tool on Ubuntu and Mint.

```
sudo ufw status                     # check current status/rules
sudo ufw enable                     # turn the firewall on
sudo ufw allow 22/tcp                # allow SSH
sudo ufw allow ssh                   # equivalent, using the service name
sudo ufw allow from 192.168.1.0/24 to any port 22   # restrict SSH to a subnet
sudo ufw deny 23                     # explicitly block a port (telnet, in this example)
sudo ufw delete allow 22/tcp         # remove a rule
```

`iptables` — the traditional, lower-level packet-filtering tool that `ufw` configures behind the scenes. Directly useful when you need fine-grained control `ufw` doesn't expose.

```
sudo iptables -L -v                 # list current rules, verbose
sudo iptables -A INPUT -p tcp --dport 22 -j ACCEPT   # allow inbound SSH
sudo iptables -A INPUT -j DROP      # drop everything else not explicitly allowed
(careful — order matters)
```

`nftables` — the modern successor to iptables, unifying IPv4/IPv6 filtering under one simpler syntax; increasingly the default backend on newer distributions even when you interact with it through `ufw` or `iptables`-compatible commands.

> ▲ **WARNING**
>
> When configuring a firewall on a machine you're accessing remotely over SSH, always double-check that your SSH rule is correctly allowed *before* enabling the firewall or adding a default-deny rule — it's easy to accidentally lock yourself out.

0.17.3 17.3 fail2ban

fail2ban monitors log files (like `/var/log/auth.log`) for repeated failed login attempts and automatically bans the offending IP address (typically via firewall rules) for a configurable period — a strong, low-effort defense against brute-force SSH attacks.

```
sudo apt install fail2ban
sudo systemctl enable --now fail2ban
sudo fail2ban-client status sshd     # check the SSH jail's current status/bans
```

0.17.4 17.4 AppArmor and SELinux

Both are **Mandatory Access Control (MAC)** systems — they add a layer of confinement *beyond* standard Unix permissions, restricting exactly what each individual application is allowed to do (which files it can read/write, which network operations it can perform), regardless of what user it's running as.

- **AppArmor** — the default MAC system on Ubuntu and Mint; profiles are relatively simple, path-based rule sets.

```
sudo aa-status                      # show loaded profiles and their mode
(enforce/complain)
sudo aa-enforce /etc/apparmor.d/usr.sbin.nginx    # switch a profile to enforcing mode
```

- **SELinux** — more granular and powerful (label-based rather than path-based), the default on Red Hat-family distributions rather than Debian-family ones. You're less likely to encounter it directly on Ubuntu/Mint/Kali, but it's worth recognizing by name.

> ✓ **TIP**
> Think of standard Unix permissions as answering "which *user* can touch this file?" and AppArmor/SELinux as answering "which *application*, doing *what specific action*, is allowed at all — even if the user running it technically has permission?" They're complementary layers, not substitutes for each other.

0.17.5 17.5 Updates, Logs, Services, Attack Surface

- **Updates** — the single highest-leverage security action available: `sudo apt update && sudo apt upgrade` regularly, and enable unattended security upgrades where appropriate (`sudo apt install unattended-upgrades`).
- **Logs** — regularly reviewing `/var/log/auth.log` and `journalctl` isn't just for troubleshooting; it's how you notice unauthorized access attempts before they become a real compromise.
- **Service minimization** — every running service is a potential entry point. Disable anything not actually needed: `systemctl list-units --type=service` to review, `systemctl disable <service>` for anything unnecessary.

- **Attack surface** — the sum of everything an attacker could potentially target: open ports, running services, installed software, user accounts. Reducing it is a constant, ongoing discipline, not a one-time setup step.

0.17.6 17.6 Hardening Checklist

✓ **TIP**

Printable Linux Hardening Checklist - [ ] System fully updated (`apt update && apt upgrade`) - [ ] Firewall enabled with a default-deny inbound policy, explicit allows only where needed - [ ] SSH: root login disabled, key-based auth only, non-default port considered - [ ] fail2ban installed and monitoring SSH - [ ] Unnecessary services disabled (`systemctl list-units --type=service`) - [ ] No unexpected SUID binaries (`find / -perm -4000 -type f 2>/dev/null`) - [ ] Strong, unique passwords for all accounts; no shared root logins - [ ] AppArmor profiles enforcing (not just "complain" mode) for exposed services - [ ] Regular log review scheduled (auth.log / journalctl) - [ ] Automatic security updates enabled where appropriate

0.17.7 17.7 Exercises

1. Check `sudo ufw status` on your system — is the firewall active? Enable it if not, allowing SSH first.
2. Install and configure `fail2ban`, then check `fail2ban-client status sshd`.
3. Check AppArmor's status with `sudo aa-status` and identify how many profiles are in enforce vs. complain mode.
4. Audit your system for SUID binaries with `find / -perm -4000 -type f 2>/dev/null` and identify three you recognize.
5. Review `journalctl -u ssh` (or `/var/log/auth.log`) for any failed login attempts.
6. Work through the Hardening Checklist above against your own VM and note any unchecked items.

0.17.8 Chapter Summary

- Least privilege is the unifying principle behind almost every Linux security practice.
- `ufw`/`iptables`/`nftables` control network traffic; `fail2ban` automates banning brute-force attackers; AppArmor/SELinux confine what individual applications can do beyond standard permissions.
- Updates, log review, and service minimization are ongoing disciplines, not one-time setup tasks.

Part I

Cybersecurity & Ethical Hacking

0.18 LINUX FOR CYBERSECURITY

0.18.1 18.1 Why Linux Dominates Security Tooling

Almost every major security tool — Nmap, Wireshark, Metasploit, Burp Suite (runs anywhere but ships in Kali), John the Ripper, Aircrack-ng — is built Linux-first. There are concrete reasons for this, not just tradition:

- **Transparency.** Open-source tools can be audited line by line. A security professional relying on a tool needs to trust it doesn't itself introduce backdoors.
- **Raw access.** Linux exposes low-level networking (raw sockets, packet crafting), the process model, and the filesystem in ways that are scriptable and composable — critical for building custom tooling quickly.
- **Server reality.** The majority of internet-facing servers run Linux, so understanding it is a prerequisite to attacking, defending, or investigating them.
- **Automation-first design.** Everything in Linux is scriptable via the shell, which matches how security work actually happens — repetitive, chainable checks across many hosts.
- **Free and legally redistributable**, allowing entire specialized distributions (Kali, Parrot OS) to bundle hundreds of tools without licensing friction.

> ❧ **SECURITY NOTE**
>
> Owning the operating system a tool runs on matters. If you don't understand what the OS is doing underneath a security tool, you can't fully trust or troubleshoot that tool's output.

0.18.2 18.2 Security Role Breakdown and Required Linux Skills

Role	What they do	Core Linux skills needed
SOC / Security Analyst	Monitor alerts, triage incidents, escalate	Logs (`journalctl`, `/var/log`), basic networking, grep/awk for log parsing, process inspection
Penetration Tester / Ethical Hacker	Simulate attacks under authorization to find weaknesses	Networking deep-dive, Nmap, web fundamentals, scripting, privilege escalation concepts
Security Engineer	Build and maintain defensive infrastructure	Firewalls, hardening, systemd, automation/scripting, package management
Incident Responder	Investigate and contain active breaches	Process/network forensics, log analysis, file timestamps, memory/disk artifacts

Role	What they do	Core Linux skills needed
Digital Forensics Analyst	Reconstruct what happened on a compromised system	Filesystem internals, hashing, `strings`/`file`, careful evidence handling (never mount write-enabled)
Red Team	Long-term authorized adversary simulation	Everything a pentester knows, plus stealth, persistence concepts, custom tooling
Blue Team	Continuous defense and detection engineering	Hardening, logging pipelines, IDS/IPS concepts, systemd/service hygiene

0.18.3 18.3 Career Path Mapping to This Book

- **SOC Analyst path:** Parts 4–11 (fundamentals) → Part 16 (logs/troubleshooting) → Part 17 (security) → Part 20 (command reference) → Part 31 (interview prep).
- **Penetration Tester path:** All fundamentals → Part 12–13 (networking/SSH) → Part 18–22 (this cybersecurity block) → Part 27–28 (labs) in an isolated environment only.
- **Security/Systems Engineer path:** Parts 8–11 (permissions/packages/processes/storage) → Part 17 (hardening) → Part 24 (advanced Linux, systemd) → Part 23 (privilege concepts, defensive framing).

◆ **OBJECTIVE**

Understand which security role you're aiming for, and use that to prioritize which later parts of this book you go deepest on first — you do not need to master every chapter equally to start being useful in a security role.

0.19 KALI LINUX FOR AUTHORIZED SECURITY TESTING

▲ **WARNING**

Everything in Parts 19–23 is written for **legal, authorized use only**: your own machines, your own isolated virtual machines, intentionally vulnerable training applications, CTF platforms, or systems where you have explicit, documented, written authorization. Accessing computers or networks you do not own or have permission to test is illegal in most countries under laws such as the U.S. Computer Fraud and Abuse Act, the UK Computer Misuse Act, and equivalent statutes elsewhere. This book will not help you target systems without authorization, and none of the material here should be interpreted as license to do so.

0.19.1 19.1 The Legal and Ethical Framework

Before touching a single tool, understand the framework professionals operate inside:

- **Authorization** — a signed, written agreement (a "Rules of Engagement" document) specifying exactly which systems may be tested, by whom, and when.
- **Scope** — the explicit boundary of what is in-bounds. Testing anything outside scope, even accidentally, is a serious breach of trust and potentially a crime.
- **Rules of Engagement (RoE)** — agreed methods, timing (e.g., outside business hours), and escalation contacts if something breaks.
- **Responsible disclosure** — if you find a real vulnerability (even on your own systems' third-party components), report it through proper channels rather than exploiting or publicizing it.

> **SECURITY NOTE**
> The single most important skill in this entire section is not a command — it's judgment. "Can I technically run this scan?" is a different question from "Am I authorized to run this scan against this target, right now?" Always answer the second question first.

0.19.2 19.2 What Kali Linux Is

Kali Linux is a Debian-based distribution maintained by Offensive Security, purpose-built for penetration testing, digital forensics, and security research. It ships with 600+ preinstalled tools organized by category, uses a rolling-release model for fast tool updates, and defaults to conventions (like easier root access in older versions) that assume the user understands the security trade-offs.

0.19.3 19.3 Tool Category Tour

Category	Purpose	Example tools
Information gathering	Discover hosts, services, and public footprint	`nmap`, `whois`, `theHarvester`
Network diagnostics	Understand connectivity and topology	`ip`, `traceroute`, `netdiscover`
Traffic analysis	Capture and inspect packets	`tcpdump`, `Wireshark`, `tshark`
Vulnerability assessment	Identify known weaknesses	`nikto`, `OpenVAS`
Web security testing	Test web app logic and input handling	`Burp Suite`, `sqlmap` (authorized targets only)
Password auditing	Test password strength on your own hashes/accounts	`John the Ripper`, `hashcat`
Wireless security concepts	Understand Wi-Fi protocol weaknesses	`aircrack-ng` suite (own hardware only)
Digital forensics	Recover and analyze evidence	`autopsy`, `binwalk`, `foremost`

Category	Purpose	Example tools
Reverse engineering	Understand compiled binaries	`ghidra`, `radare2`

This book covers the **concepts and safe, foundational commands** behind several of these categories (Parts 20–22). Deep tool-specific mastery of things like Burp Suite or Ghidra is a natural next step once these Linux and networking fundamentals are solid.

0.19.4 19.4 Why Kali Should Not Be Your First Everyday Distro

- It's tuned for testing, not daily productivity — fewer general desktop conveniences, more aggressive default tooling.
- Many tools run better and are safer to learn with root habits and network configurations that are not ideal for day-to-day browsing, email, or personal use.
- Rolling-release tooling can be less stable than an LTS distro for routine work.
- Learning fundamentals (Parts 1–17) on Ubuntu or Mint first means when you get to Kali, you're focused on security concepts — not still fighting basic Linux usage.

0.19.5 19.5 Lab-Only Usage Policy

Every hands-on exercise from here forward assumes one of these targets only:

1. Your own physical machine (with tools like `strings`, `sha256sum`, log analysis).
2. A local virtual machine you created and control.
3. An intentionally vulnerable training application (e.g., a deliberately insecure web app or VM designed for learning) running locally.
4. A CTF (Capture The Flag) platform designed for this purpose.
5. A system for which you hold explicit, written authorization.

> ★ **LAB**
> Set up your authorized lab environment now using Part 28 before running any commands from Parts 20–22 for the first time.

0.20 ESSENTIAL ETHICAL-HACKING LINUX COMMANDS

> ▲ **WARNING**
> All examples below target `127.0.0.1`, a local lab VM's private IP (e.g. `192.168.56.x` on a host-only network), or explicitly-authorized scope. Never substitute a real-world domain or IP you don't control.

0.20.1 20.1 Network Inspection

`ip` / `ss` / `ping` / `traceroute` / `arp` / `ip neigh` — covered in full in Part 12. In a security context these answer: what's my attack surface, what's reachable, what's the path to a

target.

Purpose: Confirm a lab target is alive and see open ports before deeper testing. **Syntax:** `ss -tuln` (local listening ports) · `ping -c 4 <lab-ip>` **Example:**

```
$ ss -tuln
Netid  State   Local Address:Port
tcp    LISTEN  0.0.0.0:22
tcp    LISTEN  127.0.0.1:3306
```

Interpretation: Port 22 (SSH) is reachable from any interface; port 3306 (MySQL) is bound to localhost only — not exposed to the network. **Defensive use:** Regularly audit your own listening ports; anything unexpected is worth investigating. **Lab exercise:** On your lab VM, run `ss -tuln` before and after installing a service (e.g. Apache) and compare.

0.20.2 20.2 Web/Network Requests: `curl`, `wget`

Purpose: Manually inspect HTTP responses, headers, and behavior — the starting point for web security testing. **Syntax:** `curl -I http://target` (headers only) · `curl -v http://target` (verbose, shows full exchange) **Example:**

```
$ curl -I http://localhost:8080
HTTP/1.1 200 OK
Server: Apache/2.4.52
X-Powered-By: PHP/7.4.3
```

Interpretation: The `Server` and `X-Powered-By` headers reveal software and version — useful for both attackers researching known CVEs and defenders auditing their own information disclosure. **Defensive use:** Suppress version-revealing headers in production (`ServerTokens Prod` in Apache, `expose_php = Off` in PHP). **Lab exercise:** Fetch headers from a local vulnerable-by-design web app and list every piece of information the headers reveal.

0.20.3 20.3 Packet Analysis Introduction: `tcpdump`, `tshark`

Full walkthrough is in Part 21. As a preview:

```
$ sudo tcpdump -i eth0 -n port 80
```

Captures all traffic on interface `eth0` to/from port 80, without resolving hostnames (`-n`, faster and quieter).

0.20.4 20.4 Nmap Fundamentals (Authorized Use Only)

> ▲ **WARNING**
> Port scanning a system you don't own or lack written authorization for can violate computer-crime law even if no data is accessed. Only run these against your own lab VMs.

What scanning does internally: Nmap sends crafted TCP/UDP/ICMP packets to target ports and classifies the port as open/closed/filtered based on the response (or lack of one). A SYN scan (`-sS`) sends a TCP SYN and inspects whether it gets SYN-ACK (open), RST (closed), or nothing (filtered by a firewall) — without completing the full TCP handshake.

Purpose: Discover open ports and running services on an authorized target. **Syntax:**

```
nmap <target>            # default scan of common ports
nmap -sV <target>        # version detection on open ports
nmap -O <target>         # OS fingerprinting (needs root)
nmap -p 22,80,443 <target>  # scan specific ports only
```

Example:

```
$ nmap -sV 192.168.56.101
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.4p1
80/tcp open  http    Apache httpd 2.4.52
```

Interpretation: Two services are exposed; version numbers let you cross-reference against known vulnerabilities (again, for *your own* authorized target). **Defensive use:** Run Nmap against your own infrastructure regularly to catch services you didn't intend to expose. **Lab exercise:** Scan your own lab VM with and without `-sV` and compare the extra detail version detection provides.

0.20.5 20.5 DNS Investigation: `dig`, `host`, `nslookup`

Covered fully in Part 12. In security work these reveal subdomains, mail servers, and infrastructure hints — always limit lookups to domains you own or are authorized to assess.

0.20.6 20.6 File Analysis: `file`, `strings`, `xxd`, `hexdump`

`file` — Purpose: identify true file type by content (magic bytes), not extension. **Example:** `file suspicious.bin` → `suspicious.bin: ELF 64-bit LSB executable` **Defensive use:** A `.jpg` that `file` reports as an executable is a major red flag.

`strings` — Purpose: extract human-readable text from a binary. **Syntax:** `strings -n 8 sample.bin` (only show strings ≥8 chars, cuts noise) **Interpretation:** Look for embedded URLs, IP addresses, file paths, or suspicious commands. **Lab exercise:** Run `strings`

against a known-safe compiled program on your own machine and identify library version strings.

`xxd` / `hexdump` — Purpose: view raw hex/ASCII bytes of a file, useful for spotting file signatures or hidden data. **Syntax:** `xxd file | head` · `hexdump -C file | head` **Example output:**

```
00000000  4d 5a 90 00 03 00 00 00  |MZ......|
```

`4d 5a` ("MZ") is the signature of a Windows executable — useful when `file`'s extension-based assumption might be wrong.

0.20.7 20.7 Hashing and Integrity: `sha256sum`, `sha512sum`, `md5sum`

Purpose: Verify a file hasn't been tampered with, or fingerprint malware samples/evidence. **Syntax:** `sha256sum file.iso` **Example:**

```
$ sha256sum ubuntu-24.04.iso
a1b2c3...  ubuntu-24.04.iso
```

Compare this against the hash published on the official download page.

> ❧ **SECURITY NOTE**
>
> **Why MD5 is unsuitable for modern security purposes:** MD5 is cryptographically broken — practical collision attacks (two different files producing the same hash) have been demonstrated since 2004. It's fine for quick, non-adversarial integrity checks (e.g., did a file download correctly) but must never be relied on where an adversary could deliberately craft a colliding file — password storage, code-signing, or forensic chain-of-custody. Use SHA-256 or SHA-512 for anything security-relevant.

0.20.8 20.8 Searching: `grep`, `find`, `locate`

Covered fully in Parts 5–6/14. Security use: hunting for secrets accidentally committed to files (`grep -r "password" .`), finding recently modified files after a suspected compromise (`find / -mmin -60 -type f`), or locating world-writable files (`find / -perm -002 -type f`).

0.20.9 20.9 Logs: `journalctl`, `dmesg`, `tail`, `grep`

Covered fully in Part 16. In security work, logs are often the only evidence of what happened — `journalctl -u ssh --since "1 hour ago"` to check recent authentication attempts is a common first move during incident triage.

> ★ **LAB**
>
> **Exercise:** On your lab VM, chain three of these commands together to answer: "has anyone tried to SSH into this machine in the last hour, and from where?" (Hint: `journalctl`, `grep`, and pattern matching on `Failed password`.)

0.21 WIRESHARK AND TCPDUMP

> ▲ **WARNING**
> Capture traffic only on networks and interfaces you own or are authorized to monitor. Intercepting others' network traffic without consent is illegal in most jurisdictions (wiretap-equivalent statutes).

0.21.1 21.1 Packet Capture Concepts

A **network interface** (e.g. `eth0`, `wlan0`) is your machine's connection point to a network. Normally, a network card only passes up frames addressed to it. **Promiscuous mode** tells the card to hand every frame on the segment to the OS, which is what lets tools like tcpdump see traffic beyond just your own conversations (only fully effective on hubs or when you're the traffic's endpoint/gateway — modern switched networks limit visibility to your own traffic unless you're specifically positioned, e.g. on a mirrored port).

0.21.2 21.2 Anatomy of a Packet (Layered View)

```
Ethernet Frame
 └── IP Packet (source/destination IP, TTL, protocol)
      └── TCP Segment / UDP Datagram (source/destination port, sequence numbers)
           └── Application Data (HTTP request, DNS query, TLS handshake, etc.)
```

- **Ethernet frame** — carries MAC addresses, operates on the local network segment.
- **IP packet** — carries logical addressing (source/destination IP) and routing info (TTL).
- **TCP segment** — adds ports, sequence/acknowledgment numbers, flags (SYN, ACK, FIN, RST) for reliable delivery.
- **UDP datagram** — simpler, no handshake, used by DNS queries and streaming.
- **DNS packets** — usually UDP/53, a query and response pair.
- **HTTP traffic** — plaintext, fully readable in a capture (a major reason HTTPS matters).
- **TLS-encrypted traffic** — you can see the handshake (SNI hostname, cipher negotiation) but not the encrypted payload, unless you control the decryption keys.

0.21.3 21.3 tcpdump: Commands and Filters

Purpose: Command-line packet capture — lightweight, scriptable, available almost everywhere. **Syntax:** `sudo tcpdump -i <interface> [filter expression]` **Common flags:** `-n` (don't resolve hostnames, faster) · `-c N` (capture N packets then stop) · `-w file.pcap` (save to file for later analysis in Wireshark) · `-A` (print packet contents in ASCII) **Examples:**

```
sudo tcpdump -i eth0 -n port 80              # HTTP traffic only
sudo tcpdump -i eth0 -n host 192.168.56.101  # traffic to/from one host
sudo tcpdump -i eth0 -n icmp                 # ping traffic only
```

```
sudo tcpdump -i eth0 -w capture.pcap        # save raw capture to a file
```

Expected output:

```
14:32:01.123456 IP 192.168.56.1.54321 > 192.168.56.101.80: Flags [S], seq 12345
```

Interpretation: `[S]` marks a TCP SYN packet — the first step of a handshake — going from port 54321 on the source to port 80 on the destination. **Common mistake:** Forgetting `sudo` — packet capture requires elevated privileges to access the raw interface. **Lab exercise:** Capture 20 packets of your own web browsing on your lab VM (`-c 20 port 80`) and identify the SYN, SYN-ACK, ACK handshake at the start.

0.21.4 21.4 Wireshark Walkthrough

Wireshark is the graphical counterpart to tcpdump, ideal for deep inspection:

1. Open Wireshark, select the interface to capture on.
2. Use the **display filter** bar (different syntax from tcpdump's capture filters), e.g. `http`, `dns`, `tcp.port == 443`.
3. Click any packet to see the layered breakdown (Ethernet → IP → TCP → Application) in the middle pane.
4. Right-click a TCP packet → **Follow → TCP Stream** to reconstruct an entire conversation (e.g. a full HTTP request/response) as readable text.

> ✓ **TIP**
> Capture with tcpdump on a headless lab VM (`-w capture.pcap`), then transfer the file to your main machine and open it in Wireshark's GUI for analysis — a very common real-world workflow.

0.21.5 21.5 Reading a Capture: Worked Example

Given this simplified capture of a DNS lookup followed by an HTTP request:

```
1  0.000  192.168.1.10 → 8.8.8.8      DNS  Standard query A example.com
2  0.041  8.8.8.8      → 192.168.1.10 DNS  Standard query response A 93.184.216.34
3  0.045  192.168.1.10 → 93.184.216.34 TCP  [SYN]
4  0.089  93.184.216.34 → 192.168.1.10 TCP  [SYN, ACK]
5  0.090  192.168.1.10 → 93.184.216.34 TCP  [ACK]
6  0.091  192.168.1.10 → 93.184.216.34 HTTP GET / HTTP/1.1
```

Interpretation: Packets 1–2 are the DNS resolution (name → IP). Packets 3–5 are the TCP three-way handshake establishing a connection to that IP on port 80. Packet 6 is the actual HTTP request riding on top of the now-established connection. This is the same sequence explained conceptually in Part 12.9 for `ping`, but here for a web request.

0.21.6 21.6 Exercises (Own Network Only)

1. Capture and identify a full TCP three-way handshake on your lab VM.

2. Capture a DNS query and respond to: what port is DNS on, and is it TCP or UDP by default?
3. Use “Follow TCP Stream” to reconstruct a plaintext HTTP request/response on a local test web server.
4. Compare a plaintext HTTP capture against an HTTPS capture of the same site — what is visible in each, and what is hidden?

0.22 WEB SECURITY FUNDAMENTALS

0.22.1 22.1 The HTTP Request-Response Cycle

Every web interaction is a request from a client (browser, `curl`) and a response from a server:

```
Client                              Server
  | --- GET /login HTTP/1.1 -----> |
  | <--- 200 OK + HTML page ------ |
```

- **Headers** carry metadata: `Content-Type`, `Cookie`, `Authorization`, `User-Agent`.
- **Cookies** are small pieces of data the server asks the browser to store and resend on future requests — the basis of most session management.
- **Sessions** let a stateless protocol (HTTP) feel stateful: after login, the server issues a session identifier (usually via cookie) so it can recognize you on subsequent requests without re-sending your password each time.

0.22.2 22.2 Authentication vs Authorization, Same-Origin Policy, CORS

- **Authentication (AuthN):** proving who you are (login).
- **Authorization (AuthZ):** determining what you, once identified, are allowed to do. A classic bug class — **Broken Access Control** — is when authentication works fine but authorization checks are missing or flawed (e.g., a regular user can access `/admin/deleteUser?id=5` just by guessing the URL).
- **Same-Origin Policy (SOP):** a browser security rule preventing a script loaded from one origin (domain+scheme+port) from freely reading data from a different origin.
- **CORS (Cross-Origin Resource Sharing):** a controlled, server-opt-in mechanism to relax SOP for specific trusted origins. Misconfigured CORS (e.g., reflecting any origin with credentials allowed) is itself a vulnerability.

0.22.3 22.3 Vulnerability Concepts

For each, the emphasis here is **understanding, prevention, and detection** — not exploitation instructions.

> ❧ **SECURITY NOTE**
>
> **SQL Injection (SQLi)** — occurs when user input is concatenated directly into a database query instead of being treated as data. If unsanitized input can alter a

query's logic, an attacker may read or modify data they shouldn't access. **Prevention:** parameterized queries / prepared statements, never string-concatenating user input into SQL. **Detection:** input validation testing, static analysis tools, web application firewalls as a secondary layer.

❧ **SECURITY NOTE**

Cross-Site Scripting (XSS) — occurs when user-supplied input is rendered back into a page without proper encoding, letting an attacker's script run in another user's browser session. **Prevention:** context-aware output encoding, Content-Security-Policy headers, treating all user input as untrusted when rendering HTML.

❧ **SECURITY NOTE**

Cross-Site Request Forgery (CSRF) — tricks a logged-in user's browser into submitting an unwanted request to a site they're authenticated with, exploiting the fact that cookies are sent automatically. **Prevention:** CSRF tokens tied to the user's session, checking the `Origin`/`Referer` header, `SameSite` cookie attributes.

❧ **SECURITY NOTE**

Broken Access Control — authorization checks that are missing, inconsistent, or only enforced in the UI rather than on the server. This is consistently the #1 category in OWASP's Top 10 web risks. **Prevention:** enforce authorization server-side on every request, default-deny, never trust client-side hiding of buttons/links as a security control.

❧ **SECURITY NOTE**

Authentication Weaknesses — weak password policies, no rate-limiting on login attempts, predictable session tokens, or missing multi-factor authentication. **Prevention:** rate limiting, account lockout policies, strong password/MFA requirements, cryptographically random session tokens.

❧ **SECURITY NOTE**

Security Misconfiguration — default credentials left in place, verbose error messages leaking stack traces, unnecessary services/ports exposed, missing security headers. Often the easiest category to fix and the most commonly found in real audits. **Prevention:** hardening checklists (see Part 17), removing default accounts, disabling debug/verbose modes in production.

0.22.4 22.4 OWASP Top 10 — Framing

The OWASP (Open Web Application Security Project) Top 10 is a widely-referenced, periodically updated ranking of the most common and impactful web application risks. It's not a checklist to memorize verbatim but a shared vocabulary the industry uses — worth reading directly from owasp.org as an ongoing reference alongside this chapter, since rankings and category definitions evolve between editions.

0.22.5 22.5 Safe Practice Labs

Learn these concepts hands-on only inside intentionally vulnerable, locally-hosted training applications built specifically for this purpose (the kind commonly used in security courses and CTFs) — never against real, third-party websites. Run them inside the isolated VM lab described in Part 28, on a host-only network with no exposure to the internet.

> ★ **LAB**
> **Objective:** Observe one vulnerability class end-to-end in a safe environment. **Steps:** 1) Deploy a locally-hosted vulnerable training app in your isolated lab VM. 2) Use `curl` or your browser's developer tools to inspect a request/response pair. 3) Identify which header, parameter, or missing check corresponds to the vulnerability class being demonstrated. 4) Write one sentence describing the fix a developer would apply. **Common mistake:** Treating a training app's deliberately-broken code as representative of good practice — the entire point is that it's wrong on purpose.

0.23 LINUX PRIVILEGE AND SECURITY CONCEPTS (DEFENSIVE)

This chapter teaches how administrators **audit and fix** dangerous configurations — the defensive counterpart to Part 8's permission model.

0.23.1 23.1 Root and Sudo Risk Model

Root can read, write, or execute anything on the system, bypassing normal permission checks entirely. `sudo` extends limited or full root capability to specific users for specific commands, logged for accountability. The risk model administrators think about:

- **Who has sudo, and to run what?** (`sudo -l` shows a user's allowed commands)
- **Is sudo access broader than necessary?** A user who only needs to restart one service shouldn't have unrestricted `ALL=(ALL) ALL` in `/etc/sudoers`.
- **Is every sudo use logged and reviewed?** (`journalctl _COMM=sudo` or `/var/log/auth.log`)

0.23.2 23.2 SUID/SGID Misconfiguration Risks (Audit Focus)

Recall from Part 8 that a SUID binary runs with its **owner's** privileges, not the caller's. This is legitimate for a small, well-understood set of system binaries (like `passwd`, which needs root to edit `/etc/shadow`). It becomes a risk when:

- A custom or third-party binary has SUID set unnecessarily.
- A SUID binary has an exploitable bug (buffer overflow, insecure temp-file handling) that lets a normal user leverage its elevated privilege.

> ❧ **SECURITY NOTE**
> **Audit command:** `find / -perm -4000 -type f 2>/dev/null` lists every SUID binary

on the system. A defensive admin reviews this list periodically and asks, for each entry: "Does this genuinely need to run as its owner, and is it a trusted, patched binary?" Anything unexpected — especially in a user's home directory — warrants immediate investigation.

0.23.3 23.3 PATH Hijacking Risk and Mitigation

Recall `PATH` (Part 14) is the ordered list of directories the shell searches for commands. If a writable, attacker-controlled directory appears **before** trusted system directories in a privileged user's `PATH`, a malicious file named identically to a common command (e.g. `ls`) could be executed instead of the real one.

Mitigation: never include `.` (current directory) in root's `PATH`; ensure `PATH` for privileged accounts lists only trusted, non-world-writable directories (`/usr/bin`, `/usr/sbin`, etc.) in that order.

0.23.4 23.4 Cron Job Security and Service Account Hygiene

- A cron job running as root that executes a script in a world-writable location is a privilege-escalation path: any user could edit that script.
- **Mitigation:** scripts run by cron (especially as root) should live in directories owned by root with `644`/`755` permissions, never `777`.
- Service accounts (accounts that run daemons, not humans) should have no login shell (`/usr/sbin/nologin`) and the minimum permissions their service actually needs.

0.23.5 23.5 Environment Variable Exposure Risks

Environment variables can leak sensitive data (API keys, credentials) into process listings (`ps aux` can sometimes reveal command-line arguments containing secrets) or into logs. **Mitigation:** prefer configuration files with restricted permissions or dedicated secrets-management tools over passing secrets as command-line arguments or plaintext environment variables where avoidable.

0.23.6 23.6 Linux Capabilities (Concept)

Traditionally a process either has full root power or none. **Capabilities** split root's power into discrete units (e.g. `CAP_NET_BIND_SERVICE` to bind to ports below 1024, `CAP_NET_RAW` for raw sockets used by `ping`) so a binary can be granted just the narrow privilege it needs instead of full root.

```
getcap /bin/ping
# /bin/ping cap_net_raw+ep
```

This is why modern `ping` doesn't need to run fully as root — it holds only the one capability it requires. Administrators can inspect (`getcap`) and assign (`setcap`) these narrowly instead of reaching for SUID-root.

0.23.7 23.7 Admin's Misconfiguration Audit Checklist

✓ **TIP**

- ☐ `sudo -l` reviewed for every user with sudo access — no broader than necessary
- ☐ `find / -perm -4000 -type f 2>/dev/null` reviewed — no unexpected SUID binaries
- ☐ `find / -perm -002 -type f 2>/dev/null` reviewed — no unexpected world-writable files
- ☐ Root's `PATH` contains only trusted directories, no `.`
- ☐ Cron scripts run as root are not writable by non-root users
- ☐ Service accounts have `nologin` shells and minimum required permissions
- ☐ No secrets in plaintext environment variables or command-line arguments where avoidable
- ☐ Capabilities (`getcap -r /`) reviewed for anything unexpected

★ **LAB**

Objective: Run a full misconfiguration audit on your own lab VM using the checklist above and document any findings. **Challenge:** Deliberately create one misconfiguration (e.g. a world-writable cron script) on a disposable VM, then find it using only the audit commands above.

Part II

Advanced & Expert Linux

0.24 ADVANCED LINUX

0.24.1 24.1 systemd Deep Dive

systemd is the init system and service manager on Ubuntu, Mint, and Kali. A **unit** is systemd's basic object — a service, mount point, timer, socket, or target. Unit files live in `/etc/systemd/system/` (admin-created/overridden) or `/lib/systemd/system/` (package-provided).

```
# /etc/systemd/system/myapp.service
[Unit]
Description=My Application
After=network.target

[Service]
ExecStart=/usr/local/bin/myapp
Restart=on-failure
User=myappuser

[Install]
WantedBy=multi-user.target
```

- `[Unit]` — metadata and ordering (`After=` doesn't create a dependency by itself, just ordering; use `Requires=` or `Wants=` for actual dependencies).
- `[Service]` — how to run and restart the process.
- `[Install]` — which target pulls this unit in when enabled.

Targets are systemd's equivalent of old-style runlevels: `multi-user.target` (normal multi-user, no GUI), `graphical.target` (desktop), `rescue.target` (single-user maintenance mode).

0.24.2 24.2 cron, crontab, and systemd Timers

`crontab -e` edits a user's scheduled jobs. Format:

```
# minute hour day month weekday command
30 2 * * * /usr/local/bin/backup.sh
```

Runs `backup.sh` at 2:30 AM daily. `crontab -l` lists jobs; `sudo crontab -e -u username` edits another user's crontab.

systemd **timers** are the modern alternative — a `.timer` unit paired with a `.service` unit, offering better logging (via `journalctl`) and dependency handling than cron.

0.24.3 24.3 Environment Variables at Scale

- `/etc/environment` — simple `KEY=value` pairs applied system-wide at login, no shell syntax.
- `/etc/profile.d/*.sh` — scripts sourced for all users' login shells, good for distributing PATH additions or tool-specific variables across a fleet.

0.24.4 24.4 Kernel Modules

```
lsmod                    # list currently loaded modules
sudo modprobe <module>   # load a module (and its dependencies)
sudo modprobe -r <module> # unload
sudo insmod <path.ko>    # load a module by direct file path (no dependency resolution)
sudo rmmod <module>      # unload by name (no dependency resolution)
```

`modprobe` is generally preferred over `insmod`/`rmmod` because it resolves dependencies automatically.

0.24.5 24.5 sysctl and Kernel Tuning

`sysctl` reads and writes kernel parameters at runtime, e.g.:

```
sysctl net.ipv4.ip_forward          # read current value
sudo sysctl -w net.ipv4.ip_forward=1  # enable IP forwarding temporarily
```

Persist changes in `/etc/sysctl.conf` or a file under `/etc/sysctl.d/`.

0.24.6 24.6 /proc and /sys

- `/proc` is a virtual filesystem exposing live kernel and process information. `cat /proc/cpuinfo`, `cat /proc/meminfo`, `cat /proc/<pid>/status` all read live kernel data structures, not real files on disk.
- `/sys` exposes a structured view of devices and kernel objects, often writable to change hardware/driver settings (e.g. `/sys/class/backlight/*/brightness`).

0.24.7 24.7 Device Management (udev)

`udev` is the device manager that creates/removes device nodes in `/dev` dynamically as hardware is plugged in or removed, and can trigger custom rules (`/etc/udev/rules.d/`) — e.g., always assigning the same device name to a specific USB serial adapter regardless of plug order.

0.24.8 24.8 The Boot Process

```
Power On
  │
  ▼
Firmware (BIOS or UEFI) — hardware self-test, finds boot device
  │
  ▼
Bootloader (GRUB) — presents OS choice, loads kernel + initramfs
  │
  ▼
Kernel initializes — mounts initramfs (temporary root filesystem)
```

```
  │
  ▼
initramfs loads drivers needed to mount the real root filesystem
  │
  ▼
Kernel switches to real root filesystem (pivot_root)
  │
  ▼
systemd (PID 1) starts — brings up targets, services, in dependency order
  │
  ▼
Login prompt / display manager
```

0.24.9 24.9 Shared Libraries

```
ldd /bin/ls          # list shared libraries a binary depends on
sudo ldconfig        # rebuild the shared library cache after installing new libs
```

Dynamic linking means multiple programs share one copy of a library in memory, saving disk space and RAM versus static linking each binary's own copy.

0.24.10 24.10 Linux Capabilities (Recap)

See Part 23.6 for the security framing. Administratively: `setcap cap_net_bind_service=+ep /usr/local/bin/myapp` lets a non-root binary bind to a privileged port without full root.

0.24.11 24.11 Containers, Namespaces, and cgroups (Foundations)

- **Namespaces** isolate what a process can *see* — its own PID numbering, its own network interfaces, its own mounted filesystems — even though it's running on the same shared kernel as everything else.
- **cgroups (control groups)** limit what a process can *use* — CPU shares, memory ceilings, I/O bandwidth.
- A **container** (Docker, Podman) is essentially a regular Linux process wrapped in namespaces (for isolation) and cgroups (for resource limits) — not a lightweight VM, but a fenced-off view of the same kernel.

```
Host Kernel (one kernel, shared)
 ├── Container A: own PID namespace, own network namespace, cgroup-limited to 1 CPU/512MB
 ├── Container B: own PID namespace, own network namespace, cgroup-limited to 2 CPU/1GB
 └── Host processes: full, unrestricted view
```

★ LAB

Objective: Inspect systemd, cron, and /proc on your own system. **Steps:** 1) `systemctl list-units --type=target` — identify your current boot target. 2) Add a test cron job that appends the date to a file every minute; verify with `cat`. 3) `cat /proc/meminfo | head` and identify total vs available memory. 4) `lsmod | wc -l` —

count loaded kernel modules.

0.25 EXPERT LINUX

0.25.1 25.1 Kernel Architecture Overview

Linux uses a **monolithic kernel with loadable modules**: core subsystems (scheduler, memory manager, filesystem layer, network stack) run together in kernel space for performance, while drivers and optional features can be loaded/unloaded as modules (Part 24.4) without recompiling the whole kernel.

0.25.2 25.2 Processes vs Threads, Scheduling

A **process** has its own memory address space; **threads** within a process share that address space but have their own stack and execution context. Linux implements threads as processes that share resources (via `clone()` with shared-memory flags), unifying the scheduling model — the **CFS (Completely Fair Scheduler)** decides which runnable task gets CPU time next, aiming to give each task a fair share proportional to priority (`nice` value, Part 10).

0.25.3 25.3 Virtual Memory and Paging

Every process sees its own private virtual address space; the kernel's memory management unit (MMU) translates virtual addresses to physical RAM addresses via **page tables**. Memory is managed in fixed-size **pages** (typically 4KB). When physical RAM is full, the kernel can move inactive pages to swap (Part 3.3) — freeing RAM for active work at the cost of speed if that data must be paged back in.

0.25.4 25.4 Filesystem Internals

An **inode** stores a file's metadata (permissions, owner, size, timestamps, and pointers to data blocks) — separately from the filename, which lives in the containing directory as a name→inode mapping. This is why hard links (Part 6.6) work: multiple directory entries can point to the same inode.

Journaling (used by ext4) writes a log of intended changes before applying them, so if the system crashes mid-write, the filesystem can replay or roll back the journal on next boot instead of ending up in an inconsistent state.

0.25.5 25.5 File Descriptors and System Calls

A **file descriptor** is a small integer a process uses to refer to an open file, socket, or pipe (0=stdin, 1=stdout, 2=stderr are always pre-opened). Every "file operation" a program does — `open()`, `read()`, `write()`, `close()` — is a **system call**: a controlled, defined entry point from user space into the kernel, which is the only code allowed to actually touch hardware.

```
lsof -p <pid>     # list all file descriptors a running process currently holds open
```

0.25.6 25.6 Signals and IPC

Signals (Part 10.3) are the kernel's way of async-notifying a process (SIGTERM, SIGKILL, SIGHUP). **Inter-Process Communication (IPC)** mechanisms let separate processes exchange data: - **Pipes** — one-directional byte stream between related processes (what `|` uses). - **Sockets** — bidirectional communication, usable between unrelated processes or across a network (what SSH, HTTP, and nearly all networking use). - **Shared memory** — a memory region mapped into multiple processes' address spaces for very fast data exchange, at the cost of needing explicit synchronization.

0.25.7 25.7 Namespaces and cgroups — Technical Depth

Linux supports several namespace types, each isolating a different resource: `PID` (process IDs), `NET` (network interfaces/routing), `MNT` (mount points), `UTS` (hostname), `IPC` (System V IPC objects), `USER` (UID/GID mappings). A container typically combines all of these. cgroups v2 unifies resource accounting (CPU, memory, I/O, PIDs) under a single hierarchy mounted at `/sys/fs/cgroup`.

0.25.8 25.8 Performance Tuning and Profiling Tools

`strace` — traces every system call a process makes, with arguments and return values. Essential for diagnosing "why is this program failing/hanging" at the lowest level. **Example:** `strace -c ls` shows a summary count of every syscall `ls` made.

`lsof` — lists open files (in the Unix sense: regular files, sockets, pipes, devices) system-wide or per process. `lsof -i :80` shows what's using port 80.

`vmstat` — reports virtual memory, process, CPU, and I/O statistics in one snapshot or repeated intervals. `vmstat 2 5` reports every 2 seconds, 5 times.

`iostat` — reports per-device disk I/O statistics (throughput, utilization) — key for diagnosing "is my disk the bottleneck."

`free` — reports RAM and swap usage. `free -h` for human-readable units.

`uptime` — shows how long the system has been running plus **load average** (1/5/15-minute averages of the run queue length — a rough proxy for CPU demand).

`sar` (System Activity Reporter, part of `sysstat`) — historical performance data collected over time, useful for spotting trends rather than just a single snapshot.

0.25.9 25.9 Exercises

1. `strace -c ls /` — identify the most-called system call.
2. `lsof -i` — list all current network connections your system has open.
3. `vmstat 1 5` — observe your system under a brief CPU-intensive task (e.g. `yes > /dev/null &` then kill it).

4. `free -h` — before and after opening several applications, note the change in used/available memory.
5. Install `sysstat` and run `sar -u 1 3` to sample CPU usage three times.

Part III

Internals, Labs & Reference

0.26 LINUX INTERNALS: WHAT REALLY HAPPENS

This chapter walks through exactly what the kernel and shell do underneath a handful of everyday commands. You do not need to memorize these mechanics, but understanding them once makes every future troubleshooting session faster — you'll be reasoning about a real machine instead of a black box.

0.26.1 26.1 Walkthrough: `ls`

```
Terminal captures "ls" + Enter
        |
        v
Shell (bash) tokenizes the line: command="ls", args=[]
        |
        v
Shell checks: is "ls" a builtin? No -> search $PATH directories in order
        |
        v
Found at /bin/ls (a real executable file)
        |
        v
Shell calls fork() -> creates a near-identical child process
        |
        v
Child process calls exec() -> replaces itself with /bin/ls's code
        |
        v
ls issues system calls: openat(), getdents64() (read directory entries),
close()
        |
        v
Kernel checks permissions on the directory, asks the filesystem driver
(ext4, etc.) to fetch the entries from disk/cache
        |
        v
ls formats the returned filenames and calls write() to stdout
        |
        v
Terminal displays the text
        |
        v
Child process exits; shell (the parent) reads the exit status and
shows a new prompt
```

> ✓ **TIP**
> Run `strace ls` (see Part 25) to watch this entire sequence of system calls in real time.

0.26.2 26.2 Walkthrough: `cd`

`cd` **must** be a shell **builtin**, not a separate program like `ls`. Here's why: when a program runs as a child process, changing *that child's* working directory has zero effect on the parent shell that launched it — the child simply exits and the parent's working directory is unchanged. `cd` only works because Bash implements it internally and calls the `chdir()` system call directly on itself, permanently changing the shell process's own working directory.

```
You type "cd /etc"
        |
        v
Bash recognizes "cd" as a builtin (no fork/exec happens)
        |
        v
Bash itself calls chdir("/etc")
        |
        v
Kernel validates /etc exists and is a directory you can access,
updates the shell process's current-working-directory pointer
        |
        v
Bash's own $PWD variable is updated; prompt reflects new location
```

0.26.3 26.3 Walkthrough: `cat`

```
"cat file.txt" -> fork/exec cat
        |
        v
cat calls open("file.txt", O_RDONLY) -> kernel returns a file descriptor
        |
        v
cat loops: read(fd, buffer, size) until 0 bytes returned (EOF)
        |
        v
Each chunk read is immediately passed to write(STDOUT_FILENO, buffer, size)
        |
        v
cat calls close(fd); process exits
```

`cat`'s simplicity is the point: it is a minimal example of the read-write system call loop that underlies almost every text-processing tool in Part 14.

0.26.4 26.4 Walkthrough: `ping`

```
"ping google.com"
        |
        v
ping resolves google.com via DNS (see Part 12: getaddrinfo() -> resolver
library -> /etc/resolv.conf -> DNS server query)
        |
        v
ping needs to construct a raw ICMP packet -> requires elevated privilege
(traditionally setuid-root binary, or the CAP_NET_RAW capability on
modern systems -- see Parts 8 and 24)
        |
        v
ping opens a raw socket via socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)
        |
        v
Kernel network stack builds and sends an ICMP Echo Request through the
NIC driver
        |
        v
Remote host's kernel replies with ICMP Echo Reply
        |
        v
ping's raw socket receives the reply, calculates round-trip time,
prints a line per reply
```

❧ **SECURITY NOTE**

`ping`'s need for raw socket access is a classic example of the **principle of least privilege** in action: modern Linux grants it a narrow `CAP_NET_RAW` capability instead of full root, so a bug in `ping` can't be leveraged into full system compromise. See Part 23.

0.26.5 26.5 Walkthrough: `sudo`

```
"sudo apt update"
        |
        v
sudo (itself a setuid-root binary) starts as your normal user
        |
        v
sudo consults /etc/sudoers (and /etc/sudoers.d/*) to check: is this
user allowed to run this command as root?
        |
        v
sudo calls into PAM (Pluggable Authentication Modules) to prompt for
and verify your password (unless within the timestamp cache window)
        |
        v
```

```
On success, sudo calls setuid(0)/setgid(0) -- raising its own
effective privilege to root -- then fork/exec's "apt update"
        |
        v
The child process (apt) now runs with root privileges; sudo logs the
invocation to the system log for accountability
```

> **SECURITY NOTE**
> Every successful and failed `sudo` invocation is logged (`/var/log/auth.log` on Debian-family systems). This audit trail is one of the main reasons production systems prefer `sudo` over sharing the root password directly.

0.26.6 26.6 Walkthrough: `ssh`

```
"ssh user@server"
        |
        v
DNS resolution of "server" (if not a raw IP)
        |
        v
TCP three-way handshake to port 22 (SYN, SYN-ACK, ACK)
        |
        v
SSH protocol version exchange, then a Diffie-Hellman style key
exchange negotiates a shared session key over the untrusted network
        |
        v
Client verifies the server's host key against ~/.ssh/known_hosts
(this is the "authenticity of host ... can't be established" prompt
the first time you connect)
        |
        v
Authentication phase: public-key auth (client proves possession of
the private key matching an entry in the server's authorized_keys)
or password auth
        |
        v
On success, server spawns a shell/session for the user; all further
traffic is encrypted using the negotiated session key
```

0.26.7 26.7 Walkthrough: `curl`

```
"curl https://example.com"
        |
        v
DNS resolution of example.com -> IP address
        |
```

```
        v
TCP three-way handshake to port 443
        |
        v
TLS handshake: certificate exchange, validation against trusted CAs,
negotiation of a symmetric session key
        |
        v
curl sends an HTTP request over the now-encrypted channel:
  GET / HTTP/1.1
  Host: example.com
        |
        v
Server processes the request and sends back an HTTP response
(status line, headers, body)
        |
        v
curl decrypts, parses the response, and prints the body (by default)
to stdout
        |
        v
TCP connection is closed (or kept alive, depending on headers)
```

✓ **TIP**

Add `-v` (`curl -v https://example.com`) to see every one of these stages printed to your terminal — DNS lookup, TCP connect, TLS handshake details, request headers, and response headers.

0.26.8 Chapter Summary

- Almost every command ultimately becomes a sequence of system calls that the kernel executes on your behalf.
- `cd` must be a shell builtin because a child process cannot change its parent's working directory.
- Networking commands (`ping`, `ssh`, `curl`) layer DNS resolution, then transport-level connection setup, then protocol-specific exchange, on top of raw sockets.
- `sudo` and setuid mechanisms are how Linux lets ordinary users perform privileged actions in a controlled, logged way.

0.27 PRACTICAL LABS

These 32 labs reinforce every part of the book with hands-on practice. Do them on a real system, VM, or WSL install — reading commands is not the same as running them. Unless a lab explicitly says otherwise, use your own machine or a disposable VM.

0.27.1 Lab 1 — Navigate the Filesystem

★ **LAB**

Objective: Move confidently around the filesystem using absolute and relative paths.
Requirements: Any Linux terminal (Ubuntu/Mint/Kali/WSL).
Commands: `pwd`, `cd`, `ls`
Steps: 1. Run `pwd` to see your starting location. 2. Run `cd /etc`, then `pwd` to confirm you moved. 3. Run `cd ..`, then `pwd` — you should be at `/`. 4. Run `cd ~`, then `pwd` — you should be back in your home directory. 5. Run `ls -la` and identify at least one hidden file (starts with `.`).
Expected Output: Your prompt/`pwd` output changes to match each directory; `ls -la` shows entries beginning with `.` and `..`.
Common Mistakes: Confusing `cd ..` (go up one level) with `cd -` (go to previous directory).
Challenge: Get from your home directory to `/var/log` and back to home using only relative paths (no `~`, no absolute paths except the first `cd`).

0.27.2 Lab 2 — Create Files and Directories

★ **LAB**

Objective: Practice creating a small directory structure from the command line.
Commands: `mkdir`, `touch`
Steps: 1. `mkdir -p ~/labs/lab02/{docs,scripts,data}` 2. `touch ~/labs/lab02/docs/notes.txt` 3. `touch ~/labs/lab02/scripts/{setup.sh,run.sh}` 4. `tree ~/labs/lab02` (or `ls -R` if `tree` isn't installed)
Expected Output: A tree showing `lab02/` containing three subdirectories, one with `notes.txt`, one with two `.sh` files.
Common Mistakes: Forgetting `-p` on `mkdir` when creating nested directories that don't exist yet, causing "No such file or directory" errors.
Challenge: Recreate the same structure using a single `mkdir -p` command with brace expansion for all three subdirectories in one line.

0.27.3 Lab 3 — File Manipulation

★ **LAB**

Objective: Copy, move, rename, and delete files safely.
Commands: `cp`, `mv`, `rm`
Steps: 1. `cp ~/labs/lab02/docs/notes.txt ~/labs/lab02/docs/notes_backup.txt` 2. `mv ~/labs/lab02/docs/notes_backup.txt ~/labs/lab02/data/` 3. `rm ~/labs/lab02/data/notes_backup.txt` 4. Confirm deletion with `ls ~/labs/lab02/data/`
Expected Output: Empty `data/` directory after step 3-4.
Common Mistakes: Using `mv` when you meant `cp` (the original disappears); forgetting `rm` has no undo/Recycle Bin.

Challenge: Use `cp -r` to duplicate the entire `~/labs/lab02` directory as `~/labs/lab02_copy` in one command.

0.27.4 Lab 4 — Permissions Repair Scenario

★ LAB

Objective: Diagnose and fix a broken permission scenario.
Commands: `chmod`, `ls -l`
Steps: 1. `touch ~/labs/script_test.sh && echo 'echo hello' >> ~/labs/script_test.sh` 2. Try to run it: `~/labs/script_test.sh` — it will fail with "Permission denied". 3. Run `ls -l ~/labs/script_test.sh` and identify the missing execute bit. 4. Fix it: `chmod u+x ~/labs/script_test.sh` 5. Run it again: `~/labs/script_test.sh` — it should print "hello".
Expected Output: "Permission denied" before the fix, "hello" after.
Common Mistakes: Using `chmod 777` as a reflexive fix — it works but grants far more access than needed. Prefer the minimum permission that solves the problem (`u+x` here).
Challenge: Set the file to be executable by the owner and group, but not readable by others (`750`), and verify with `ls -l`.

0.27.5 Lab 5 — Users and Groups Management

★ LAB

Objective: Create a user, add them to a group, and verify.
Commands: `sudo useradd`, `sudo passwd`, `sudo usermod`, `id`, `groups`
Steps: 1. `sudo useradd -m labuser` 2. `sudo passwd labuser` (set a temporary password) 3. `sudo groupadd labteam` 4. `sudo usermod -aG labteam labuser` 5. `id labuser` — confirm `labteam` appears in the group list.
Expected Output: `id labuser` output includes `groups=...,labteam`.
Common Mistakes: Using `-G` alone without `-a` on `usermod`, which *replaces* all of a user's supplementary groups instead of adding one.
Challenge: Remove `labuser` from `labteam` without deleting the user, using `gpasswd -d`.

0.27.6 Lab 6 — Process Management

★ LAB

Objective: Start, observe, and safely terminate a runaway process.
Commands: `ps`, `top`, `kill`, `jobs`, `bg`, `fg`
Steps: 1. Run `sleep 300 &` — starts a background process for 5 minutes. 2. Run `jobs` — see it listed. 3. Run `ps aux | grep sleep` and note the PID. 4. Run `kill <PID>` (SIGTERM) — it should stop. 5. Repeat, but this time run `kill -9 <PID>` and note the difference in what SIGTERM vs SIGKILL allows.

Expected Output: The process disappears from `ps aux` after either kill.
Common Mistakes: Reaching for `kill -9` by default — it skips cleanup, which can corrupt state for processes mid-write.
Challenge: Start three `sleep` processes at once and kill all of them in a single command using `pkill sleep`.

0.27.7 Lab 7 — Package Installation and Removal

★ **LAB**
Objective: Practice the full APT workflow.
Commands: `apt update`, `apt install`, `apt remove`, `apt purge`, `apt search`
Steps: 1. `sudo apt update` 2. `apt search cowsay` 3. `sudo apt install cowsay` 4. `cowsay "Linux Mastery"` 5. `sudo apt remove cowsay` then `sudo apt purge cowsay` — note the difference (remove keeps config files, purge deletes them too).
Expected Output: ASCII cow says the message in step 4; package is gone after step 5.
Common Mistakes: Forgetting `sudo apt update` before installing on a freshly-installed system, leading to "unable to locate package" errors from a stale package index.
Challenge: Use `apt list --installed | wc -l` before and after to confirm the package count changed by exactly one.

0.27.8 Lab 8 — Managing a systemd Service

★ **LAB**
Objective: Control a real service's lifecycle with `systemctl`.
Commands: `systemctl status/start/stop/restart/enable/disable`, `journalctl -u`
Steps: 1. `sudo apt install apache2` (or any lightweight service, e.g. `openssh-server` if already using SSH). 2. `sudo systemctl status apache2` 3. `sudo systemctl stop apache2` then check status again. 4. `sudo systemctl start apache2` 5. `sudo systemctl enable apache2` — confirm it will start at boot. 6. `journalctl -u apache2 --since "10 min ago"`
Expected Output: Status shows `active (running)` after start, `inactive (dead)` after stop.
Common Mistakes: Confusing `enable` (start at boot) with `start` (start right now) — they are independent.
Challenge: Disable the service again and confirm with `systemctl is-enabled apache2`.

0.27.9 Lab 9 — Networking Basics

★ **LAB**

Objective: Inspect and test your machine's network configuration.
Commands: `ip addr`, `ip route`, `ping`
Steps: 1. `ip addr show` — identify your machine's IP address and interface name. 2. `ip route show` — identify your default gateway. 3. `ping -c 4 8.8.8.8` — test raw internet connectivity (bypassing DNS). 4. `ping -c 4 google.com` — test DNS + connectivity together.
Expected Output: Both pings succeed with reasonable round-trip times if you have internet access.
Common Mistakes: Assuming a failed `ping google.com` but successful `ping 8.8.8.8` means "no internet" — it usually means a DNS problem specifically, not a connectivity problem. See Lab 10.
Challenge: Identify your MAC address using `ip link show`.

0.27.10 Lab 10 — DNS Troubleshooting Scenario

★ **LAB**

Objective: Diagnose a DNS-specific failure.
Commands: `dig`, `nslookup`, `cat /etc/resolv.conf`
Steps: 1. `dig example.com` — inspect the ANSWER SECTION. 2. `dig example.com @1.1.1.1` — force a specific DNS server and compare. 3. `cat /etc/resolv.conf` — check which resolver your system uses by default. 4. If `dig example.com` fails but `dig example.com @1.1.1.1` succeeds, your configured resolver is the problem.
Expected Output: Both `dig` commands return an ANSWER SECTION with an IP address.
Common Mistakes: Editing `/etc/resolv.conf` directly on systems managed by NetworkManager or systemd-resolved — changes get overwritten. Use `nmcli` or the resolved config instead (Part 12).
Challenge: Use `dig +trace example.com` to watch the full recursive resolution from root servers downward.

0.27.11 Lab 11 — SSH Key-Based Login Setup

★ **LAB**

Objective: Set up passwordless SSH authentication between two local VMs.
Requirements: Two VMs on the same host-only/NAT network (see Part 28), or one VM plus your host.
Commands: `ssh-keygen`, `ssh-copy-id`, `ssh`
Steps: 1. On the client VM: `ssh-keygen -t ed25519 -C "lab-key"` (press Enter through the prompts for a lab key). 2. `ssh-copy-id user@<server-ip>` 3. `ssh user@<server-ip>` — you should log in without a password prompt. 4. On the server, inspect `~/.ssh/authorized_keys` to see the copied public key.

Expected Output: Step 3 logs in with no password prompt.
Common Mistakes: Copying the **private** key instead of the public key (`.pub`) to the server — never do this; the private key must never leave the client.
Challenge: Disable password authentication entirely in `/etc/ssh/sshd_config` (`PasswordAuthentication no`) and restart `sshd`, confirming only key-based login works.

0.27.12 Lab 12 — Bash Scripting: First Script

★ **LAB**

Objective: Write, make executable, and run your first script.
Steps: 1. Create `~/labs/hello.sh`:

```
#!/bin/bash
echo "Hello, $(whoami). Today is $(date +%A)."
```

2. `chmod +x ~/labs/hello.sh`
3. `./hello.sh`

Expected Output: A greeting with your username and the current day of the week.
Common Mistakes: Forgetting the shebang line (`#!/bin/bash`) — the script may run under the wrong shell and behave unexpectedly.
Challenge: Modify the script to accept a name as `$1` and greet that name instead of `whoami`, with a fallback to `whoami` if no argument is given.

0.27.13 Lab 13 — Log Analysis

★ **LAB**

Objective: Find a specific error using `journalctl` and `grep`.
Commands: `journalctl`, `grep`
Steps: 1. `journalctl -p err -b` — show only error-level+ messages from this boot. 2. `journalctl -u ssh --since "1 hour ago" | grep -i fail` (adjust unit name/timeframe as needed). 3. Identify the timestamp and process name of any error found.
Expected Output: Zero or more log lines with `err`/`crit` priority; a grep result showing matching lines if any exist.
Common Mistakes: Forgetting `-b` scopes to the *current* boot only — use `-b -1` for the previous boot when debugging a crash-reboot.
Challenge: Export the last hour of logs to a file with `journalctl --since "1 hour ago" > ~/labs/last_hour.log` and grep it offline.

0.27.14 Lab 14 — Firewall Configuration with ufw

★ **LAB**

Objective: Configure a basic firewall policy.
Commands: `ufw`
Steps: 1. `sudo ufw status` (should be inactive by default on most systems). 2. `sudo ufw default deny incoming` 3. `sudo ufw default allow outgoing` 4. `sudo ufw allow ssh` (critical — do this *before* enabling if you're on a remote VM over SSH). 5. `sudo ufw enable` 6. `sudo ufw status verbose`
Expected Output: Status shows "Status: active" with SSH allowed, everything else denied inbound.

▲ **WARNING**

If you are connected to this machine **over SSH**, enabling `ufw` without first allowing SSH will lock you out immediately. Always run `sudo ufw allow ssh` before `sudo ufw enable` on a remote box.

Challenge: Add a rule allowing HTTP (port 80) only from your local subnet, e.g. `sudo ufw allow from 192.168.1.0/24 to any port 80`.

0.27.15 Lab 15 — Packet Capture with tcpdump

★ **LAB**

Objective: Capture and read live traffic on your own network.
Requirements: Your own machine/network only.
Commands: `tcpdump`
Steps: 1. `sudo tcpdump -i any icmp -c 5` 2. In another terminal, run `ping -c 5 8.8.8.8` 3. Observe the ICMP echo request/reply pairs captured. 4. Try `sudo tcpdump -i any port 53 -c 5` while running `dig example.com` in another terminal.
Expected Output: Captured packet lines showing source/destination IPs and protocol details.
Common Mistakes: Forgetting `sudo` — packet capture requires elevated privilege (`CAP_NET_RAW`).
Challenge: Capture and save to a file with `-w capture.pcap`, then open it in Wireshark for GUI-based inspection (see Part 21).

0.27.16 Lab 16 — Disk Usage Investigation and Cleanup

★ **LAB**

Objective: Find what's consuming disk space.
Commands: `df -h`, `du -sh`, `du -h --max-depth=1`
Steps: 1. `df -h` — identify which filesystem is nearly full (if any). 2. `du -h --max-depth=1 /var | sort -rh | head` — find the largest top-level subdirectories of `/var`. 3. Drill into the largest one with another `du -h --max-depth=1`.

Expected Output: A sorted list of directory sizes, largest first.
Common Mistakes: Running `du -sh /` without narrowing scope first — it works but is slow and gives you one big number instead of an actionable breakdown.
Challenge: Find and safely clear old package cache with `sudo apt clean` and measure the space freed via `df -h` before/after.

0.27.17 Lab 17 — Archiving and Backup with tar

★ **LAB**
Objective: Create and restore a compressed backup.
Commands: `tar`
Steps: 1. `tar -czvf ~/labs/backup.tar.gz ~/labs/lab02` 2. `mkdir ~/labs/restore_test` 3. `tar -xzvf ~/labs/backup.tar.gz -C ~/labs/restore_test` 4. `diff -r ~/labs/lab02 ~/labs/restore_test/labs/lab02` — should show no differences.
Expected Output: `diff -r` produces no output (identical directories).
Common Mistakes: Mixing up `-c` (create) and `-x` (extract) flags, or forgetting `-C` to specify an extraction target directory (extracting into your current directory unexpectedly).
Challenge: Repeat using `zip`/`unzip` instead of `tar`/`gzip` and compare resulting file sizes.

0.27.18 Lab 18 — Cron Job Scheduling

★ **LAB**
Objective: Schedule a recurring task.
Commands: `crontab -e`, `crontab -l`
Steps: 1. `crontab -e` 2. Add: `* * * * * echo "$(date)" >> ~/labs/cron_test.log` 3. Wait 2–3 minutes. 4. `cat ~/labs/cron_test.log` — confirm new lines appear roughly once a minute. 5. Remove the line and save to stop it.
Expected Output: A growing log file with one timestamp per minute.
Common Mistakes: Assuming cron jobs run with your full interactive shell environment — cron uses a minimal environment, so scripts relying on your `PATH` or aliases may fail (always use absolute paths in cron jobs).
Challenge: Schedule a job to run only on weekdays at 9 AM: `0 9 * * 1-5 ...`.

0.27.19 Lab 19 — Vim Editing Challenge

★ **LAB**
Objective: Practice core Vim motions and modes without leaving the keyboard's home row.
Steps: 1. `vim ~/labs/vim_practice.txt` 2. Press `i`, type five lines of any text, press `Esc`. 3. Save without quitting: `:w` 4. Move to the top with `gg`, delete the first line with

`dd`. 5. Search for a word you typed using `/word`, press `n` to jump to the next match if repeated. 6. Save and quit: `:wq`
Expected Output: The file on disk reflects your edits; `dd` removed exactly one line.
Common Mistakes: Typing text while still in Normal mode (before pressing `i`), which triggers unexpected commands instead of inserting text.
Challenge: Use visual mode (`v`, then arrow keys, then `d`) to delete a specific chunk of text mid-line.

0.27.20 Lab 20 — Symbolic vs. Hard Link Demo

★ **LAB**
Objective: See the practical difference between link types.
Commands: `ln`, `ln -s`, `ls -li`
Steps: 1. `echo "original" > ~/labs/original.txt` 2. `ln ~/labs/original.txt ~/labs/hardlink.txt` (hard link) 3. `ln -s ~/labs/original.txt ~/labs/symlink.txt` (symbolic link) 4. `ls -li ~/labs/*.txt` — note the hard link shares the same inode number as the original; the symlink has a different inode and shows an arrow (`->`) to its target. 5. `rm ~/labs/original.txt` — the hard link still works; the symlink is now broken.
Expected Output: `cat ~/labs/hardlink.txt` still shows "original" after deletion; `cat ~/labs/symlink.txt` errors with "No such file or directory".
Common Mistakes: Assuming symlinks and hard links behave identically — they diverge exactly at the moment the original file is deleted.
Challenge: Create a symlink to a directory and confirm `cd` follows it transparently.

0.27.21 Lab 21 — File Integrity Check with sha256sum

★ **LAB**
Objective: Verify file integrity using cryptographic hashing.
Commands: `sha256sum`
Steps: 1. `echo "important data" > ~/labs/file.txt` 2. `sha256sum ~/labs/file.txt > ~/labs/file.sha256` 3. `sha256sum -c ~/labs/file.sha256` — should report OK. 4. `echo "tampered" >> ~/labs/file.txt` 5. `sha256sum -c ~/labs/file.sha256` — should now report FAILED.
Expected Output: "OK" before tampering, "FAILED" after.
Common Mistakes: Using MD5 for anything security-relevant — MD5 collisions are computationally feasible, meaning a malicious file can be crafted to match a known MD5 hash. Always prefer SHA-256 or better.
Challenge: Download any file with `wget`, and verify it against a checksum published on the vendor's website (a real-world integrity check).

0.27.22 Lab 22 — Local Web Server + curl Testing

★ **LAB**

Objective: Stand up a minimal web server and inspect it with curl.
Commands: `python3 -m http.server`, `curl`
Steps: 1. `cd ~/labs/lab02 && python3 -m http.server 8080` 2. In another terminal: `curl -v http://localhost:8080/` 3. Observe the request and response headers in the verbose output. 4. `curl -I http://localhost:8080/` — headers only.
Expected Output: An HTTP 200 response with a directory listing body, and visible `Server`/`Content-Type` headers.
Common Mistakes: Forgetting the server is only bound to your own machine by default in this simple form — accessible from other devices only if you bind to `0.0.0.0` and your firewall allows it.
Challenge: Time the request with `curl -o /dev/null -s -w "%{time_total}\n" http://localhost:8080/`.

0.27.23 Lab 23 — Authorized Nmap Scan Against Your Own VM

★ **LAB**

Objective: Run a basic Nmap scan against a machine you own, on an isolated lab network.
Requirements: Your own local VM (see Part 28). **Never point this at a machine you don't own or lack written authorization to test.**
Commands: `nmap`
Steps: 1. Confirm the target VM's IP: `ip addr` on the target. 2. From the Kali VM: `nmap <target-ip>` 3. `nmap -sV <target-ip>` — add service/version detection. 4. `nmap -p 22,80,443 <target-ip>` — scan specific ports only.
Expected Output: A list of open ports and, with `-sV`, guessed service names/versions.
Common Mistakes: Running scans against IP ranges you don't control "just to see what's out there" — this can violate laws and terms of service even on networks you're merely connected to (e.g., public Wi-Fi, university networks, cloud providers).
Challenge: Compare results of a scan before and after stopping a service (e.g., `sudo systemctl stop apache2`) on the target.

0.27.24 Lab 24 — Safe File Analysis of an Unknown Binary

★ **LAB**

Objective: Practice static analysis techniques on a harmless local binary.
Commands: `file`, `strings`, `sha256sum`
Steps: 1. Pick any local system binary, e.g. `/bin/ls`. 2. `file /bin/ls` — identify architecture and type. 3. `strings /bin/ls | less` — scan for readable text, embedded paths, or library names. 4. `sha256sum /bin/ls` — record its hash for later comparison.

Expected Output: `file` reports something like "ELF 64-bit LSB pie executable"; `strings` reveals shared library names and usage text.
Common Mistakes: Assuming `strings` output is exhaustive analysis — it only surfaces plain ASCII/Unicode text, not the binary's actual logic. Real reverse engineering (Part 19) needs disassemblers/debuggers.
Challenge: Compare `file` output for a `.txt`, a `.jpg`, and a compiled binary to see how the tool identifies each by content, not extension.

0.27.25 Lab 25 — Reading /proc for a Running Process

★ **LAB**
Objective: Inspect a live process's kernel-exposed metadata.
Commands: `/proc/<pid>/`
Steps: 1. `sleep 300 &` then note the PID (`echo $!`). 2. `cat /proc/<pid>/status | head -20` — memory usage, state, UID. 3. `cat /proc/<pid>/cmdline | tr '\0' ' '` — the exact command line that launched it. 4. `ls -l /proc/<pid>/fd` — open file descriptors.
Expected Output: Structured process metadata pulled directly from kernel memory, no special tools required.
Common Mistakes: Forgetting `/proc` entries disappear the instant the process exits — capture what you need while it's running.
Challenge: Find the same process's info using `ps` and `top` and confirm the numbers match what you read from `/proc` directly.

0.27.26 Lab 26 — Building a Bash System Health-Check Script

★ **LAB**
Objective: Combine Part 15 concepts into a working diagnostic tool.
Steps: 1. Write a script that reports: hostname, uptime, disk usage of `/`, memory usage, and the top 3 CPU-consuming processes. 2. Use `hostname`, `uptime`, `df -h /`, `free -h`, and `ps aux --sort=-%cpu | head -4`. 3. Make it executable and run it.
Expected Output: A clean multi-section report printed to the terminal.
Common Mistakes: Not quoting variables (`$var` vs `"$var"`), which breaks on output containing spaces.
Challenge: Add a threshold check that prints a warning line if disk usage on `/` exceeds 80%.

0.27.27 Lab 27 — Setting Up fail2ban on SSH

★ **LAB**
Objective: Add automated brute-force protection to SSH.
Commands: `apt install fail2ban`, `fail2ban-client`

Steps: 1. `sudo apt install fail2ban` 2. `sudo systemctl status fail2ban` — confirm it's running with the default `sshd` jail active. 3. `sudo fail2ban-client status sshd` 4. From another machine (or deliberately, on your lab VM only), fail an SSH login several times in a row. 5. `sudo fail2ban-client status sshd` again — the offending IP should now appear as banned.
Expected Output: The attacking IP listed under "Banned IP list" after repeated failures.
Common Mistakes: Testing this against a machine you rely on for remote access without a console/snapshot fallback — you can lock yourself out. Always test on a disposable lab VM with console access.
Challenge: Unban your own test IP with `sudo fail2ban-client set sshd unbanip <ip>`.

0.27.28 Lab 28 — AppArmor Profile Inspection

★ **LAB**
Objective: See AppArmor's confinement in action.
Commands: `aa-status`, `sudo aa-enforce`, `sudo aa-complain`
Steps: 1. `sudo aa-status` — list loaded profiles and their mode (enforce/complain). 2. Pick a profile in "enforce" mode for an installed application. 3. `sudo aa-complain /path/to/profile` — switch it to complain mode (logs violations but doesn't block them). 4. `sudo aa-enforce /path/to/profile` — switch back.
Expected Output: `aa-status` output changes to reflect the mode switch.
Common Mistakes: Disabling AppArmor entirely to "fix" an application error, instead of reading `/var/log/syslog` (or `journalctl`) for the specific denial and adjusting the profile.
Challenge: Find one AppArmor denial in your logs (`grep apparmor /var/log/syslog`) and explain in your own words what it blocked and why.

0.27.29 Lab 29 — Recovering from a Full Disk

★ **LAB**
Objective: Simulate and recover from a disk-full scenario safely.
Steps: 1. On a disposable VM, create a large dummy file: `fallocate -l 2G ~/labs/bigfile.img` (adjust size relative to your VM's disk). 2. `df -h` — observe usage climbing. 3. Try installing a package — observe the "No space left on device" error. 4. `rm ~/labs/bigfile.img` 5. `df -h` — confirm space is reclaimed; retry the install.
Expected Output: Install fails while the disk is full, succeeds after cleanup.
Common Mistakes: Panicking and deleting system files indiscriminately — always identify *what's* consuming space (Lab 16) before deleting anything.
Challenge: Find and clear a real, safe-to-remove source of disk usage on your VM, such as old kernel packages (`sudo apt autoremove`) or package cache (`sudo apt clean`).

0.27.30 Lab 30 — Mini Capture-the-Flag on a Local Vulnerable VM

★ **LAB**

Objective: Apply reconnaissance, enumeration, and basic exploitation concepts against an intentionally vulnerable target, entirely offline in your own lab.
Requirements: A downloaded vulnerable-by-design VM (see Part 28 for legal sourcing) imported into VirtualBox/VMware on a host-only network.
Steps: 1. Identify the target's IP on your isolated lab network. 2. Run an Nmap scan to enumerate open ports and services (Lab 23). 3. Investigate any web service found with a browser and `curl`. 4. Follow the specific vulnerable VM's documented walkthrough hints (most training VMs ship with objectives) to find the first flag. 5. Document every command you ran and why, in a lab notebook (Part 28.6).
Expected Output: A discovered "flag" file or string, per the target VM's design.
Common Mistakes: Skipping documentation — in real engagements, the write-up is often as valuable as the finding itself.

0.27.31 Lab 31 (Bonus) — Building a Two-VM Isolated Pentest Lab from Scratch

★ **LAB**

Objective: Construct the lab environment described in Part 28 yourself, end to end.
Steps: 1. Install VirtualBox (or VMware). 2. Create a **Host-only** or **internal** network in the hypervisor's network manager. 3. Import/install Kali Linux as VM 1, attached only to that isolated network. 4. Import a vulnerable training VM as VM 2, attached to the same isolated network, with no bridged/NAT-to-internet adapter. 5. Confirm the two VMs can ping each other but that VM 2 cannot reach the public internet. 6. Take a snapshot of both VMs in this clean state.
Expected Output: Two VMs that can talk to each other and nothing else.
Common Mistakes: Leaving a bridged adapter attached "just in case" — this defeats the isolation that makes the lab safe to experiment in.

0.27.32 Lab 32 (Bonus) — Wireshark Capture + Protocol Identification Challenge

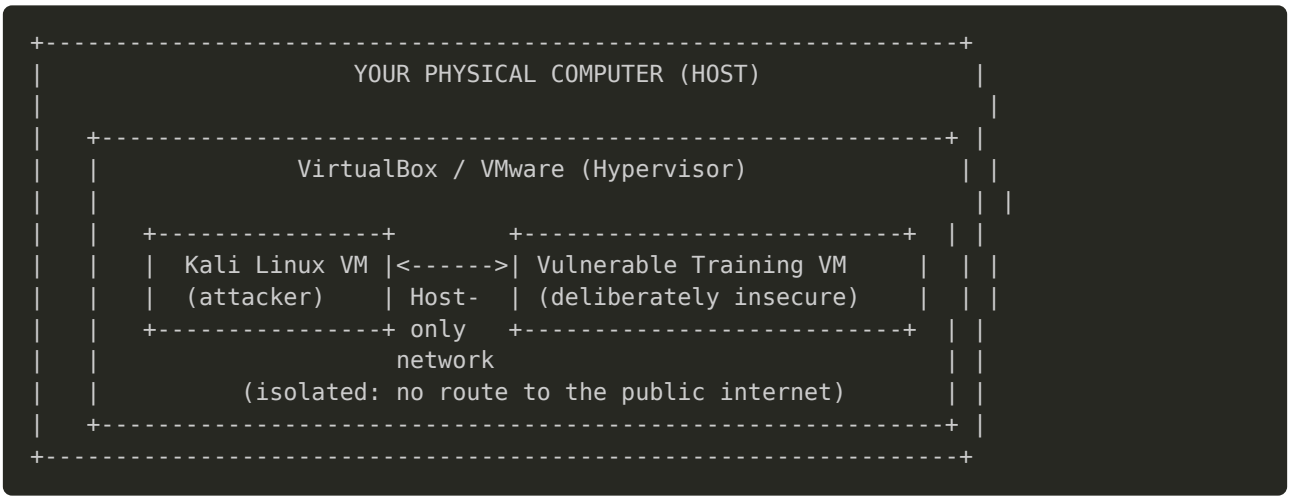
★ **LAB**

Objective: Capture and correctly identify five different protocols in one sitting.
Steps: 1. Open Wireshark on an interface with traffic (or on your lab network). 2. Generate varied traffic: `ping`, `dig`, browsing to an HTTP (not HTTPS) test site, an SSH connection, and a local file transfer via `scp`. 3. Apply display filters one at a time: `icmp`, `dns`, `http`, `ssh`, `tcp`. 4. For each, identify: source/destination, port number, and one field unique to that protocol (e.g., DNS query name, HTTP method).
Expected Output: Five correctly filtered and identified protocol captures.
Common Mistakes: Capturing on the wrong interface (e.g., a virtual adapter with no real traffic) and concluding "nothing is happening" — always confirm you're cap-

turing on the interface actually carrying your traffic.

0.28 SAFE CYBERSECURITY LAB

0.28.1 28.1 Lab Architecture


YOUR PHYSICAL COMPUTER (HOST)
VirtualBox / VMware (Hypervisor)
Kali Linux VM (attacker)
Host-only network
Vulnerable Training VM (deliberately insecure)
(isolated: no route to the public internet)

This two-VM pattern — one attacker machine, one deliberately vulnerable target, both isolated on a private virtual network — is the standard, legal way to practice offensive security skills. Everything you do stays contained on your own hardware.

◆ **OBJECTIVE**

Build an isolated environment where you can practice reconnaissance, scanning, and exploitation concepts without any risk of touching a network or system you don't own.

0.28.2 28.2 NAT vs. Host-Only Networking

Mode	What it does	Use it when
NAT	VM shares the host's internet connection but is not directly reachable from outside	You need the VM to download updates/tools
Bridged	VM appears as its own device on your physical network	You need the VM to be reachable by other real devices (rarely needed for this book)
Host-only	VM can talk only to the host and other host-only VMs — no internet access at all	Running attack/target pairs safely — **this is what the lab above uses**

Mode	What it does	Use it when
Internal Network	Similar to host-only but excludes even the host — VMs can only see each other	Maximum isolation for target VMs that should never reach anything else

❧ **SECURITY NOTE**

For the attacker VM (Kali) and any vulnerable target VM, use **Host-only** or **Internal** networking for the pair, so they can reach each other but nothing else. If Kali also needs internet access to update tools, give it a *second* adapter on NAT — but never put the vulnerable target VM on that same NAT adapter.

0.28.3 28.3 Snapshots

A **snapshot** captures a VM's exact disk and memory state so you can return to it instantly.

Workflow: 1. Install and fully configure your VM (Kali or a target). 2. Take a snapshot labeled something like `clean-install`. 3. Experiment freely — break things, install tools, misconfigure services on purpose. 4. When you want a fresh start, **revert to snapshot** instead of reinstalling from scratch.

✓ **TIP**

Snapshot *before* every lab exercise that modifies the system. This turns any mistake into a thirty-second fix instead of a full reinstall.

0.28.4 28.4 Why Isolation Matters — Legally and Technically

- **Legally:** Running scanning or exploitation tools against systems you don't own or lack explicit written authorization for can violate computer-crime laws (e.g., the Computer Fraud and Abuse Act in the US, and equivalent laws elsewhere), even if you "were just testing" or "didn't cause damage."
- **Technically:** Vulnerable training VMs are, by design, full of security holes. An isolated network ensures that if one gets compromised (by you, intentionally, as part of learning) it can't become a pivot point into your real network or the internet.
- **Practically:** Isolation also protects you from accidentally affecting production systems, shared networks (like your home Wi-Fi with other household devices), or misconfiguring something that leaks your target VM to the internet.

0.28.5 28.5 Resetting Machines Safely

- Prefer **snapshot revert** over manual cleanup — manual cleanup risks missing a change.
- If a VM's network configuration was altered during a lab, verify host-only/internal settings are still correct *after* reverting.

- Periodically re-verify isolation: from the target VM, attempt `ping 8.8.8.8` — it should fail if isolation is configured correctly.

0.28.6 28.6 Documenting Experiments

Keep a simple lab notebook (a plain text file or markdown file is fine) for every session:

```
## Session: 2026-07-29

Target: metasploitable-vm (192.168.56.20)
Attacker: kali-vm (192.168.56.10)

1. nmap -sV 192.168.56.20
   -> Found: 21/tcp open ftp, 22/tcp open ssh, 80/tcp open http
2. Investigated port 80 with curl -v
   -> Found outdated CMS version in response headers
3. Findings: outdated software version disclosed in HTTP headers
   -> Real-world fix: suppress version banners, patch software
```

> ✓ **TIP**
> Writing down *why* you ran each command, not just the command itself, is exactly the habit that separates a professional penetration-testing report from a random list of terminal history.

0.28.7 28.7 Legal Training Environments

Categories of legitimate places to practice (always follow each platform's specific rules of engagement):

- **Intentionally vulnerable VMs** — downloadable virtual machines built specifically to be attacked in a lab, covering web apps, network services, and privilege-escalation scenarios.
- **CTF (Capture the Flag) platforms** — hosted or downloadable challenges designed for legal practice, ranging from beginner to expert difficulty.
- **Web-application security trainers** — deliberately vulnerable web apps meant to be run locally to practice concepts like those in Part 22 (SQL injection, XSS, etc.).
- **Vendor-sanctioned bug bounty programs** — real-world systems where the owner has explicitly authorized security testing under a defined scope; always read and follow the scope document exactly.

> ▲ **WARNING**
> Scanning or attacking anything outside an environment you own or are explicitly authorized to test — including "just checking" a neighbor's Wi-Fi, a public website, or a friend's server without written permission — is not covered by any of the above and can carry serious legal consequences.

0.28.8 Chapter Summary

- Use a two-VM host-only/internal network setup: one attacker VM, one vulnerable target, isolated from the internet and your real network.
- Snapshots make experimentation safe and instantly reversible.
- Isolation exists for both legal and technical safety reasons.
- Document what you do and why — it builds the habit that professional security work requires.
- Only practice against systems you own, isolated lab VMs, CTFs, or explicitly authorized targets.

Part IV

Quick Reference

0.29 COMMAND CHEAT SHEETS

Concise, printable references. Full explanations are in the parts noted next to each title.

0.29.1 Linux Basics (Part 4)

Command	Purpose
`pwd`	Print working directory
`ls -la`	List all files, long format
`cd path`	Change directory
`mkdir -p dir`	Create directory (with parents)
`touch file`	Create empty file / update timestamp
`cp -r src dst`	Copy (recursively for directories)
`mv src dst`	Move or rename
`rm -r dir`	Remove (recursively)
`cat file`	Print file contents
`history`	Show command history
`clear`	Clear terminal screen
`whoami`	Show current user
`uname -a`	Kernel/system info
`reboot` / `shutdown now`	Restart / power off

0.29.2 Filesystem (Part 5)

Command	Purpose
`tree`	Visual directory tree
`find /path -name "*.log"`	Search by name
`locate filename`	Fast indexed search
`which cmd`	Show which executable runs for `cmd`
`file name`	Identify file type
`stat file`	Detailed metadata
`du -sh dir`	Directory size
`df -h`	Filesystem free space

0.29.3 Permissions (Part 8)

Command	Purpose
`ls -l`	View permissions
`chmod 755 file`	Set permissions numerically
`chmod u+x file`	Add execute for owner
`chown user:group file`	Change owner/group
`umask`	View/set default permission mask

Command	Purpose
`chmod u+s file`	Set SUID
`chmod g+s dir`	Set SGID
`chmod +t dir`	Set sticky bit

0.29.4 Users (Part 8)

Command	Purpose
`id`	Show UID/GID/groups
`who` / `w`	Logged-in users
`sudo useradd -m name`	Create user with home dir
`sudo usermod -aG group user`	Add user to group
`sudo passwd user`	Set/change password
`groups user`	List a user's groups
`su - user`	Switch user

0.29.5 Processes (Part 10)

Command	Purpose
`ps aux`	List all processes
`top` / `htop`	Live process monitor
`kill PID`	Send SIGTERM
`kill -9 PID`	Send SIGKILL (force)
`pkill name`	Kill by process name
`jobs`	List shell jobs
`bg` / `fg`	Resume job in background/foreground
`nohup cmd &`	Run immune to hangup
`systemctl status svc`	Service status

0.29.6 Networking (Part 12)

Command	Purpose
`ip addr`	Show IP addresses
`ip route`	Show routing table
`ping -c4 host`	Test reachability
`ss -tulnp`	Show listening ports
`traceroute host`	Show route hops
`dig domain`	DNS lookup (detailed)
`curl -v url`	HTTP request with detail
`nmcli device status`	NetworkManager device state

0.29.7 Package Management (Part 9)

Command	Purpose
`sudo apt update`	Refresh package index
`sudo apt upgrade`	Upgrade installed packages
`sudo apt install pkg`	Install package
`sudo apt remove pkg`	Remove (keep config)
`sudo apt purge pkg`	Remove + config
`apt search term`	Search repositories
`dpkg -l`	List installed .deb packages
`snap list`	List installed snaps

0.29.8 systemd (Part 10, 24)

Command	Purpose
`systemctl start/stop/restart svc`	Control a service now
`systemctl enable/disable svc`	Control boot-time behavior
`systemctl status svc`	Current state
`journalctl -u svc`	Logs for a unit
`journalctl -f`	Follow logs live
`systemctl list-units --type=target`	Boot targets

0.29.9 SSH (Part 13)

Command	Purpose
`ssh user@host`	Connect
`ssh-keygen -t ed25519`	Generate key pair
`ssh-copy-id user@host`	Install public key remotely
`scp file user@host:path`	Copy file over SSH
`sftp user@host`	Interactive secure file transfer

0.29.10 Bash & Text Processing (Part 14)

Command	Purpose
`grep -i "text" file`	Case-insensitive search
`grep -r "text" dir`	Recursive search
`cut -d, -f2 file`	Extract column 2 (comma-delimited)
`sort file` / `sort -n`	Sort lines / numerically
`uniq -c`	Count duplicate lines
`wc -l file`	Count lines
`sed 's/old/new/g' file`	Find/replace

Command	Purpose
`awk '{print $1}' file`	Print first field
`xargs`	Build commands from input
`tee file`	Write to file and stdout simultaneously

0.29.11 Logs (Part 16)

Command	Purpose
`journalctl -b`	Logs from current boot
`journalctl -p err`	Error-level and above
`journalctl -u name --since "1 hour ago"`	Scoped by unit and time
`dmesg | tail`	Recent kernel messages
`tail -f /var/log/syslog`	Live-follow a log file

0.29.12 Security (Part 17, 23)

Command	Purpose
`sudo ufw enable`	Turn on firewall
`sudo ufw allow 22/tcp`	Allow a port
`sudo ufw status verbose`	Firewall rule summary
`sudo fail2ban-client status sshd`	Brute-force protection status
`sudo aa-status`	AppArmor profile status
`find / -perm -4000 2>/dev/null`	Find SUID binaries

0.29.13 Kali Linux / Ethical-Hacking Lab Commands (Parts 19–21)

Command	Purpose
`nmap target`	Basic port scan
`nmap -sV target`	Service/version detection
`nmap -p 22,80,443 target`	Scan specific ports
`tcpdump -i any -c 10`	Capture 10 packets
`tshark -i any`	Terminal-based Wireshark
`sha256sum file`	Integrity hash
`strings binary`	Extract readable text from a binary
`whois domain`	Registration lookup

0.29.14 Troubleshooting (Part 16)

Symptom	First Commands
Disk full	`df -h`, `du -sh --max-depth=1`
High CPU	`top`, `ps aux --sort=-%cpu`
High memory	`free -h`, `top`
Network down	`ip addr`, `ping 8.8.8.8`, `ping domain`
DNS issue	`dig domain`, `cat /etc/resolv.conf`
Service won't start	`systemctl status svc`, `journalctl -u svc`
Permission denied	`ls -l`, `id`, `whoami`

0.30 DANGEROUS COMMANDS

The commands in this chapter are powerful, legitimate, and necessary — but each can destroy data or make a system unbootable in seconds if used carelessly. Read every warning before you ever type these on a machine that matters.

> ▲ **WARNING**
>
> **Before you press Enter on any command in this chapter, ask: "Do I know exactly what every argument does, and could I recover if this is wrong?"** If the answer to either half is no, stop and re-check, or practice on a disposable VM first.

0.30.1 30.1 `rm -rf`

What it does: Recursively (`-r`) and forcibly (`-f`, no confirmation prompts) deletes files and directories.

Why it's dangerous: There is no Recycle Bin. `rm -rf /some/path` permanently deletes everything under that path, instantly, with no built-in undo.

```
rm -rf /important/data      # gone, permanently, no confirmation
```

Safer alternative: Use `rm -ri` (interactive, prompts per file) while learning, double-check the path with `pwd`/`ls` first, and consider `trash-cli` for a Recycle-Bin-like workflow on personal machines.

> ▲ **WARNING**
>
> Never run `rm -rf /` or `rm -rf *` from an unexpected directory. Modern `rm` refuses `rm -rf /` by default, but `rm -rf ./*` or a wrong variable expansion in a script (`rm -rf "$dir"/*` where `$dir` is accidentally empty, deleting from `/`) has destroyed real systems.

0.30.2 30.2 `dd`

What it does: Low-level block-by-block copying, often used to write disk images (`dd if=input.iso of=/dev/sdX`).

Why it's dangerous: `dd` has no safety checks. If `of=` points at the wrong device, it silently overwrites that device's contents — including your main hard drive — with no warning and no progress bar by default.

```
sudo dd if=ubuntu.iso of=/dev/sdb bs=4M status=progress   # correct: writes to a USB
drive
sudo dd if=ubuntu.iso of=/dev/sda bs=4M status=progress   # WRONG: /dev/sda may be your
main disk
```

Safer alternative: Always run `lsblk` immediately before a `dd` command to confirm the exact device name, and prefer GUI tools (Balena Etcher, Rufus) for routine USB writing.

0.30.3 30.3 `mkfs`

What it does: Formats a partition with a new filesystem (`mkfs.ext4`, `mkfs.vfat`, etc.), destroying all existing data on it.

```
sudo mkfs.ext4 /dev/sdb1     # erases everything on that partition
```

Safer alternative: Confirm the target with `lsblk` and `blkid` first; never run against a partition you haven't explicitly identified.

0.30.4 30.4 `fdisk` / `parted`

What they do: Create, delete, and resize disk partitions.

Why they're dangerous: Deleting or resizing the wrong partition can destroy an entire operating system installation or all user data on that disk, and mistakes here are a common cause of failed dual-boot installations (see Part 3).

Safer alternative: Always back up first; use `lsblk`/`fdisk -l` to triple-check the device path; practice on a VM's virtual disk before ever touching a real dual-boot machine.

0.30.5 30.5 `chmod -R` / `chown -R`

What they do: Recursively change permissions or ownership of every file under a directory.

Why they're dangerous: `chmod -R 777 /` (even by typo) makes the entire system world-writable, a catastrophic security hole; `chown -R` on the wrong path can break a system by handing ownership of critical files to the wrong user.

```
sudo chmod -R 777 /            # catastrophic — never run this
sudo chown -R www-data:www-data /var/www/mysite   # correct, scoped use
```

Safer alternative: Always scope `-R` operations to the narrowest directory possible, and use `find ... -exec chmod ...` when you need to target only files or only directories, not both indiscriminately.

0.30.6 30.6 `shutdown` / `reboot`

What they do: Power off or restart the machine.

Why caution matters: On a remote server, an unplanned `reboot` or a mistimed `shutdown now` can end your only session and, if a service fails to come back up automatically, take a production system offline.

Safer practice: Use `shutdown -r +5 "Rebooting for maintenance in 5 minutes"` to give warning and a cancel window (`shutdown -c`), and always confirm you have out-of-band (console) access to a remote machine before rebooting it.

0.30.7 30.7 `kill -9`

What it does: Sends SIGKILL, which terminates a process immediately with no chance to clean up (close files, flush writes, release locks).

Why it's risky: A database or application killed with `-9` mid-write can leave corrupted files or inconsistent state behind. Always try plain `kill` (SIGTERM) first and give the process a few seconds to exit gracefully.

```
kill PID       # ask nicely (SIGTERM) — try this first
kill -9 PID    # force it — last resort only
```

0.30.8 30.8 Pre-Enter Checklist

✓ **TIP**

Before running any command from this chapter:

1. Re-read the full command, argument by argument.
2. Confirm the target path/device with a read-only command first (`pwd`, `ls`, `lsblk`).
3. Ask: is there a backup or snapshot if this goes wrong?
4. If practicing to learn, do it in a disposable VM — never on a machine holding data you care about.

0.30.9 Chapter Summary

- `rm -rf`, `dd`, `mkfs`, `fdisk`/`parted`, `chmod -R`/`chown -R`, `shutdown`/`reboot`, and `kill -9` are all legitimate, necessary tools — the risk is in *targeting* them incorrectly, not in the commands themselves.
- Always verify the target (path, device, PID) with a read-only command before running a destructive one.
- Practice destructive commands only in disposable VMs, never on systems holding data you care about.

0.31 INTERVIEW PREPARATION

0.31.1 31.1 Beginner Questions

Q1. What is the difference between the Linux kernel and a Linux distribution? A: The kernel is the core software that manages hardware (CPU, memory, devices) and exposes system calls. A distribution packages the kernel with GNU tools, a package manager, and other software into a complete, installable OS (e.g., Ubuntu, Mint, Kali).

Q2. What does `chmod 755 file` do? A: Sets permissions to `rwxr-xr-x` — owner gets read/write/execute (7), group and others get read/execute (5 each). Common for scripts and executables that should be run but not modified by non-owners.

Q3. What's the difference between `>` and `>>` in Bash? A: `>` redirects output to a file, overwriting it. `>>` appends to the end of the file, preserving existing content.

Q4. How do you find out which directory you're currently in? A: `pwd` (Print Working Directory).

Q5. What is the purpose of `sudo`? A: Runs a single command with elevated (root) privileges, after authenticating, rather than logging in as root directly — this preserves an audit trail and follows the principle of least privilege.

Q6. What's the difference between `cp` and `mv`? A: `cp` duplicates a file (original remains); `mv` moves/renames it (original is gone from its old location).

0.31.2 31.2 Intermediate Questions

Q7. Explain the difference between a hard link and a symbolic link. A: A hard link is a second directory entry pointing to the same inode (same underlying data) — deleting the original doesn't affect it. A symbolic link is a separate file containing a path to the target — deleting the original breaks it (a "dangling" symlink).

Q8. What happens when you run `kill -9` on a process versus a plain `kill`? A: Plain `kill` sends SIGTERM, asking the process to terminate gracefully (it can catch the signal and clean up). `kill -9` sends SIGKILL, which the kernel enforces immediately — the process cannot intercept or ignore it, and gets no chance to clean up.

Q9. What's stored in `/etc/passwd` vs. `/etc/shadow`? A: `/etc/passwd` holds user account metadata (username, UID, GID, home directory, shell) and is world-readable. `/etc/shadow` holds the actual hashed passwords and password-aging data, and is readable only by root, for security.

Q10. What does `systemctl enable` do differently from `systemctl start`? A: `enable` configures a service to start automatically at boot (creates symlinks in the appropriate target's `.wants` directory); `start` starts it immediately, once, with no effect on boot behavior. They're independent — you can do either, both, or neither.

Q11. What is the difference between a process and a thread? A: A process is an independent execution context with its own memory space; a thread is a unit of execution within a process, sharing that process's memory with other threads in the same process.

Q12. What does `/etc/fstab` control? A: It defines which filesystems get mounted automatically at boot, including their device, mount point, filesystem type, and mount

options.

0.31.3 31.3 Advanced Questions

Q13. Walk through what happens, at a system-call level, when you run `cat file.txt`. A: The shell forks and execs `cat`. `cat` calls `open()` on the file to get a file descriptor, then loops calling `read()` into a buffer and `write()` of that buffer to stdout, until `read()` returns 0 (EOF), then calls `close()` and exits.

Q14. Why must `cd` be a shell builtin rather than a standalone program? A: If `cd` were a separate executable, it would run as a child process; changing that child's working directory has no effect on the parent shell once the child exits. Only a builtin, executing inside the shell's own process, can change the shell's own working directory via `chdir()`.

Q15. Explain the difference between SUID, SGID, and the sticky bit. A: SUID makes an executable run with the *file owner's* privileges regardless of who runs it (e.g., `passwd` runs as root so any user can update the shadow file safely through a controlled interface). SGID on an executable runs it with the *group owner's* privileges; on a directory, new files inherit the directory's group. The sticky bit on a shared directory (like `/tmp`) restricts deletion of a file to its owner (or root), even though everyone can write to the directory.

Q16. What is the difference between `ufw`, `iptables`, and `nftables`? A: `iptables` is the traditional Linux firewall framework (built on `netfilter`); `nftables` is its modern successor with a cleaner syntax and better performance, now the default backend on most distributions. `ufw` ("Uncomplicated Firewall") is a simplified front-end that generates `iptables`/`nftables` rules from simple commands like `ufw allow 22`.

Q17. What's the difference between a namespace and a cgroup in the context of containers? A: A **namespace** isolates *what a process can see* (its own view of process IDs, network interfaces, mount points, hostname, etc.). A **cgroup** (control group) limits and accounts for *how much of a resource* a process or group of processes can use (CPU, memory, I/O). Containers combine both: namespaces for isolation, cgroups for resource limits.

0.31.4 31.4 System Administrator Questions

Q18. A disk is reporting "No space left on device" but `df -h` shows free space. What could be wrong? A: The filesystem may have run out of **inodes** rather than disk blocks — check with `df -i`. This happens when a huge number of small files (e.g., session files, mail queue files) exhaust the inode table even though raw byte capacity remains.

Q19. A service fails to start after a config change. What's your troubleshooting sequence? A: `systemctl status service` for the immediate error, then `journalctl -u service -n 50` for detailed logs, verify the config file syntax if the service has a `--test`/`-t` flag (e.g., `nginx -t`), and check file permissions on any files the service needs to read.

Q20. How would you find which process is listening on port 8080? A: `sudo ss -tulnp | grep 8080`, or `sudo lsof -i :8080`.

Q21. Describe your process for safely applying updates to a production Linux server. A: Review release notes/changelogs for breaking changes, snapshot or back up the system, apply updates during a maintenance window, test critical services afterward, and have a rollback plan (snapshot revert or documented downgrade steps) ready before starting.

0.31.5 31.5 Cybersecurity-Focused Questions

Q22. What is the principle of least privilege, and give a Linux-specific example. A: Granting only the minimum access needed to perform a task. Example: giving a web server process its own dedicated low-privilege user (e.g., `www-data`) rather than running it as root, so a compromise of the web application doesn't hand over full system control.

Q23. Why is MD5 considered unsuitable for security purposes today? A: MD5 is cryptographically broken — practical collision attacks exist, meaning two different inputs can be crafted to produce the same hash. This makes it unsuitable for integrity verification or password storage; SHA-256 or stronger (and for passwords, a dedicated slow hash like bcrypt/Argon2) should be used instead.

Q24. What does `nmap -sV target` do, and why does it require authorization? A: It performs service and version detection on open ports, actively probing them to identify running software and versions. Because it sends unsolicited probe traffic to the target, running it against systems you don't own or lack explicit permission for can violate computer-crime laws, even without causing damage.

Q25. What's the difference between SIEM-style log analysis and the raw `journalctl//var/log` review covered in this book? A: This book's approach (grep/journalctl/manual review) works for single-host troubleshooting and small labs. A SIEM aggregates logs from many hosts centrally, correlates events across systems, and applies detection rules at scale — the underlying log data and Linux logging mechanisms are the same; a SIEM adds centralization and correlation on top.

Q26. How would you detect a SUID binary that shouldn't have that bit set? A: `find / -perm -4000 -type f 2>/dev/null` lists all SUID binaries system-wide; compare the output against a known-good baseline (or common legitimate SUID binaries like `passwd`, `sudo`, `mount`) to spot anomalies added by an attacker for persistence/privilege escalation.

0.31.6 31.6 Scenario / Troubleshooting Questions

Scenario A: *A user reports they can't SSH into a server that was working yesterday. Walk through your diagnosis.* A: Check basic connectivity first (`ping` the server), then whether SSH's port is reachable (`nc -zv host 22` or `telnet host 22`), then check the SSH service status on the server itself if you have console/other access (`systemctl status sshd`), review `/var/log/auth.log` for failed attempts or configuration errors, and check firewall rules (`ufw status`) for recent changes that might block port 22.

Scenario B: *A web application is intermittently slow. What Linux-level checks do you run before assuming it's an application bug?* A: `top`/`htop` for CPU and memory pressure, `free -h` for memory exhaustion/swap thrashing, `iostat` for disk I/O bottlenecks, `ss -s` for

connection counts, and `journalctl`/application logs for errors coinciding with the slow periods.

Scenario C: *You find a cron job owned by a non-admin user that runs a script from `/tmp` every minute. What's your assessment?* A: This is a red flag for possible persistence established by an attacker — `/tmp` is world-writable, so any user (or compromised process) could have placed that script there. Investigate the script's contents, its modification timestamp, the user's other recent activity, and treat it as a potential incident until proven otherwise rather than simply deleting it (preserve evidence first).

Scenario D: *A colleague ran `chmod -R 777 /var/www` to "fix a permissions error" and it worked, but you're concerned. Why, and what should be done instead?* A: `777` grants read/write/execute to everyone, including any other user or process on the system — a serious security risk for a web-accessible directory (arbitrary file write can lead to full compromise). The right fix is to identify the *actual* narrow permission problem (usually wrong ownership, not permissions) and correct it with something like `chown -R www-data:www-data /var/www` and `chmod -R 750`, scoped to only what the web server process needs.

Part V

Roadmaps & Master Reference

0.32 LEARNING ROADMAP

Use whichever roadmap matches your available time. Each day assumes roughly 45–90 minutes of focused study and practice. Skipping a day is fine — resume where you left off rather than restarting.

0.32.1 32.1 30-Day Linux Roadmap (Fundamentals Track)

Day	Topic	Practice / Lab	Outcome
1	What is Linux, kernel vs distro (Part 1)	Read + self-check questions	Explain Linux vs a distro in your own words
2	Ubuntu / Mint / Kali overview (Part 2)	Compare table review	Choose which distro to install
3	Installation planning (Part 3)	Download ISO, prepare USB	Bootable installer ready
4	Install in a VM	Full VirtualBox install	Working Linux VM
5	First day basics (Part 4)	`pwd`,`ls`,`cd`,`mkdir`	Comfortable navigating
6	File management (Part 6)	Lab 2, Lab 3	Create/copy/move/delete confidently
7	Filesystem hierarchy (Part 5)	Explore `/etc`,`/var`,`/usr`	Know what each top-level dir is for
8	Reading/editing files (Part 7)	Vim basics, Lab 19	Edit a file end-to-end in Vim
9	Users & permissions part 1 (Part 8)	`chmod`,`ls -l`	Read any permission string
10	Users & permissions part 2 (Part 8)	Lab 4, Lab 5	Create a user, fix a permission bug
11	Package management (Part 9)	Lab 7	Install/remove software via APT
12	Processes (Part 10)	Lab 6	Kill a runaway process safely
13	systemd basics (Part 10)	Lab 8	Start/stop/enable a service
14	Storage (Part 11)	Lab 16	Diagnose disk usage
15	Review + catch-up day	Redo any failed lab	Consolidate weeks 1–2
16	Networking fundamentals (Part 12)	Lab 9	Explain your own IP/gateway setup

Day	Topic	Practice / Lab	Outcome
17	DNS deep dive (Part 12)	Lab 10	Diagnose a DNS-specific failure
18	SSH (Part 13)	Lab 11	Passwordless SSH between two VMs
19	Bash shell basics (Part 14)	Pipes/redirects practice	Chain 3 commands with a pipe
20	Text processing tools (Part 14)	`grep`,`sed`,`awk` practice	Extract data from a log file
21	Bash scripting basics (Part 15)	Lab 12	Write and run your first script
22	Bash scripting: loops/conditionals (Part 15)	Project 1 (system info script)	A working diagnostic script
23	Logs & troubleshooting (Part 16)	Lab 13	Find an error in journalctl
24	Linux security fundamentals (Part 17)	Lab 14 (ufw)	Firewall configured correctly
25	Dangerous commands (Part 30)	Read carefully, no VM edits needed	Know what never to run carelessly
26	Cheat sheets review (Part 29)	Skim all tables	Recognize every command at a glance
27	Mock interview: beginner + intermediate (Part 31)	Answer without looking	Self-assess weak areas
28	Revisit weakest topic from day 27	Targeted re-study	Close the gap
29	Full review day	Re-run 3 favorite labs from memory	Confidence check
30	Reflection + plan next 30 days	Decide: sysadmin path or security path	Clear direction forward

0.32.2 32.2 60-Day Linux Roadmap (Fundamentals + Advanced)

Days 1–30 are identical to the 30-Day Roadmap above. Continue with:

Day	Topic	Practice / Lab	Outcome
31–32	Bash scripting projects 2–4 (Part 15)	Disk monitor, backup script	Two more working scripts

Day	Topic	Practice / Lab	Outcome
33–34	Bash scripting projects 5–7 (Part 15)	Network diagnostic, service checker	Complete script portfolio
35	Archiving & backups (Part 6)	Lab 17	Reliable tar backup/restore
36	Cron scheduling (Part 24)	Lab 18	Recurring job configured correctly
37	Advanced permissions: SUID/SGID/sticky (Part 8)	`find -perm -4000` audit	Explain every SUID binary on your system
38	ACLs (Part 8)	`getfacl`/`setfacl` practice	Grant access beyond owner/group/other
39	Advanced networking (Part 12)	`ss`,`traceroute` deep dive	Diagnose a routing issue
40	Wireshark & tcpdump intro (Part 21)	Lab 15	Capture and read a packet exchange
41	systemd deep dive (Part 24)	Unit file inspection	Explain a unit file’s dependencies
42	Kernel modules & /proc, /sys (Part 24)	Lab from Part 24	Read live kernel state
43	Boot process (Part 24)	Diagram walkthrough	Explain BIOS→GRUB→kernel→systemd
44	Containers, namespaces, cgroups intro (Part 24)	Read + diagram	Explain what a container isolates
45	Expert Linux: processes/threads/memory (Part 25)	Read + self-check	Explain virtual memory basics
46	Performance tools: strace, lsof, vmstat (Part 25)	Lab from Part 25	Profile a running process
47	Linux internals walkthroughs (Part 26)	Trace `ls` and `ssh` yourself	Explain any command’s system-call path
48	Review: Advanced Linux	Redo 2 failed labs	Consolidate weeks 5–7
49	Linux security hardening (Part 17)	Full hardening checklist	Harden your lab VM end-to-end
50	fail2ban + AppArmor (Part 17)	Lab 27, Lab 28	Working intrusion protection

Day	Topic	Practice / Lab	Outcome
51	Privilege/security concepts, defensive (Part 23)	Lab from Part 23	Run a misconfiguration audit
52	Interview prep: advanced + sysadmin (Part 31)	Answer without looking	Self-assess weak areas
53	Mock troubleshooting scenarios (Part 31.6)	Solve 4 scenarios cold	Build diagnostic speed
54–56	Build a personal home-lab project	Your choice (e.g., host a service, harden it, monitor it)	A portfolio-worthy project
57	Master command reference review (Part 33)	Skim, flag unfamiliar commands	Fewer than 10 unfamiliar commands remain
58	Full review day	Re-run 5 favorite labs from memory	Confidence check
59	Documentation day	Write up your home-lab project (Part 28.6 style)	A shareable write-up
60	Reflection + decide next 30 days	Choose: deepen sysadmin or move into Part 33 security track	Clear direction forward

0.32.3 32.3 90-Day Linux + Cybersecurity Roadmap

Days 1–60 are identical to the 60-Day Roadmap above. Continue with the cybersecurity-focused block:

Day	Topic	Practice / Lab	Outcome
61	Linux for cybersecurity roles (Part 18)	Read + map to a target role	Pick a target security role to aim for
62	Kali Linux responsibly (Part 19)	Read legal/ethical framework carefully	Understand authorized-use boundaries
63	Build the isolated lab (Part 28)	Lab 31	Two-VM host-only lab ready
64	Network inspection commands (Part 20)	Practice on lab VM	Comfortable with `ss`, `ip`, `arp`

Day	Topic	Practice / Lab	Outcome
65	curl/wget for recon (Part 20)	Practice on lab VM	Extract headers/version info from a service
66	Nmap fundamentals (Part 20)	Lab 23	Run and interpret a service-version scan
67	DNS investigation tools (Part 20)	`dig`,`host`,`nslookup` practice	Enumerate DNS records for a lab domain
68	File analysis: file/strings/xxd/hexdump (Part 20)	Lab 24	Statically inspect an unknown binary
69	Hashing & integrity (Part 20)	Lab 21	Detect tampering via SHA-256
70	tcpdump deep dive (Part 21)	Lab 15 again, more filters	Filter captures by host/port/protocol
71	Wireshark deep dive (Part 21)	Lab 32	Identify 5 protocols in one capture session
72	Web security fundamentals part 1 (Part 22)	Read: HTTP, headers, cookies, sessions	Explain a request/response cycle fully
73	Web security fundamentals part 2 (Part 22)	Read: SQLi, XSS, CSRF concepts	Explain each vulnerability class in plain language
74	Safe web-app lab exercise (Part 22)	Deploy vulnerable training app locally	Observe one vulnerability class end-to-end
75	Privilege escalation concepts, defensive framing (Part 23)	Lab from Part 23	Explain how misconfigurations get discovered
76	Mini CTF attempt 1 (Part 27, Lab 30)	Full walkthrough, take notes	First flag captured with documentation
77	Mini CTF attempt 2	New vulnerable VM, fresh notes	Faster enumeration than attempt 1
78	Digital forensics concepts (Part 18)	Read + `/proc` inspection lab	Explain evidence-handling basics
79	Reverse engineering concepts (Part 19)	`strings`/`file` on sample binaries	Comfortable with static analysis vocabulary
80	Review: offensive fundamentals	Redo Nmap + tcpdump labs from memory	Consolidate weeks 12–13

Day	Topic	Practice / Lab	Outcome
81	Cybersecurity interview prep (Part 31.5)	Answer without looking	Self-assess weak areas
82	Scenario-based troubleshooting drills (Part 31.6)	Solve cold, time yourself	Build real-world diagnostic speed
83–85	Full lab project: document a complete mini-assessment	Recon → scan → finding → write-up on a training VM	A portfolio-worthy security write-up
86	Legal/ethical review (Part 19, Part 28.7)	Re-read authorization boundaries	Zero ambiguity on what's authorized
87	Master command reference: security section (Part 33)	Skim, flag unfamiliar commands	Fewer than 5 unfamiliar security commands remain
88	Resume/portfolio prep	Compile your labs, scripts, and write-ups	A presentable project list
89	Full review day	Re-run 5 favorite labs across the whole book from memory	Confidence check
90	Reflection + next steps	Decide: CTF platforms, certifications, or job applications	A concrete next action, not just "keep learning"

✓ **TIP**

These roadmaps are deliberately realistic, not aggressive. If a day takes you two sessions instead of one, that's normal — consistency across 30/60/90 days matters far more than speed on any single day.

0.33 COMMAND MASTER REFERENCE

A categorized reference of 150+ commands used throughout this book. "Opt" shows the most important flags, not every possible option — see each command's own `man` page for the complete list.

0.33.1 Navigation & Files

Command	Meaning	Key Options	Example	Purpose
`pwd`	Print working directory		`pwd`	Show current location
`ls`	List directory contents	`-l -a -h -R`	`ls -la`	View files, permissions, hidden files
`cd`	Change directory		`cd /etc`	Navigate
`tree`	Show directory tree	`-L n`	`tree -L 2`	Visualize structure
`mkdir`	Make directory	`-p`	`mkdir -p a/b/c`	Create nested dirs
`rmdir`	Remove empty directory		`rmdir empty/`	Delete empty dir
`touch`	Create empty file / update timestamp		`touch f.txt`	New file
`cp`	Copy	`-r -v -i`	`cp -r src dst`	Duplicate files/dirs
`mv`	Move / rename		`mv old new`	Rename or relocate
`rm`	Remove	`-r -f -i`	`rm -r dir`	Delete files/dirs
`ln`	Create link	`-s`	`ln -s target link`	Hard/symbolic links
`find`	Search filesystem	`-name -type -perm`	`find / -name "*.log"`	Locate files by criteria
`locate`	Fast indexed search		`locate nginx.conf`	Quick filename search
`which`	Show executable path		`which python3`	Find which binary runs
`whereis`	Locate bi-nary/man/source		`whereis ls`	Broader binary lookup
`file`	Identify file type		`file image.png`	Determine content type
`stat`	Detailed file metadata		`stat file`	Inode, timestamps, size
`du`	Disk usage of files/dirs	`-s -h`	`du -sh dir`	Directory size
`df`	Filesystem free space	`-h`	`df -h`	Free/used space per mount

0.33.2 Archiving & Compression

Command	Meaning	Key Options	Example	Purpose
`tar`	Archive files	`-c -x -z -v -f`	`tar -czvf a.tar.gz dir/`	Bundle + compress
`gzip`	Compress a file	`-d` (decompress)	`gzip file`	.gz compression
`gunzip`	Decompress .gz		`gunzip file.gz`	Reverse gzip
`zip`	Create zip archive	`-r`	`zip -r a.zip dir/`	Cross-platform archive
`unzip`	Extract zip archive		`unzip a.zip`	Reverse zip

0.33.3 Viewing & Editing Files

Command	Meaning	Key Options	Example	Purpose
`cat`	Print whole file		`cat file`	Dump contents
`less`	Paginated viewer		`less file`	Scroll through large files
`more`	Simple pager		`more file`	Basic pagination
`head`	Show first lines	`-n`	`head -n 20 file`	Preview file start
`tail`	Show last lines	`-n -f`	`tail -f log`	Preview end / live-follow
`nano`	Simple text editor		`nano file`	Beginner-friendly editing
`vim`	Modal text editor		`vim file`	Powerful keyboard-driven editing

0.33.4 Users, Groups & Permissions

Command	Meaning	Key Options	Example	Purpose
`whoami`	Current username		`whoami`	Identify current user
`id`	UID/GID/groups		`id`	Full identity info
`who`	Logged-in users		`who`	Session list
`w`	Logged-in users + activity		`w`	Detailed session info

Command	Meaning	Key Options	Example	Purpose
`users`	List logged-in usernames		`users`	Simple user list
`useradd`	Create user (low-level)	`-m`	`sudo useradd -m bob`	Add account + home dir
`adduser`	Create user (interactive, Debian)		`sudo adduser bob`	Friendlier user creation
`usermod`	Modify user	`-aG`	`sudo usermod -aG sudo bob`	Change account/groups
`userdel`	Delete user	`-r`	`sudo userdel -r bob`	Remove account + home
`groupadd`	Create group		`sudo groupadd devs`	New group
`groups`	Show a user's groups		`groups bob`	Group membership
`passwd`	Change password		`passwd`	Update login password
`su`	Switch user	`-`	`su - bob`	Become another user
`sudo`	Run as another user (root)	`-u`	`sudo apt update`	Elevated single command
`chmod`	Change permissions	`-R` numeric/symbolic	`chmod 755 file`	Set r/w/x bits
`chown`	Change owner	`-R`	`chown bob:devs file`	Set owner/group
`chgrp`	Change group only	`-R`	`chgrp devs file`	Set group ownership
`umask`	Default permission mask		`umask 022`	Control default new-file perms
`getfacl`	View ACLs		`getfacl file`	Extended permissions view
`setfacl`	Set ACLs	`-m`	`setfacl -m u:bob:rw file`	Grant beyond owner/group/other

0.33.5 Package Management

Command	Meaning	Key Options	Example	Purpose
`apt update`	Refresh package index		`sudo apt update`	Sync repo metadata

Command	Meaning	Key Options	Example	Purpose
`apt upgrade`	Upgrade installed packages		`sudo apt upgrade`	Apply newer versions
`apt install`	Install package		`sudo apt install nginx`	Add software
`apt remove`	Remove package		`sudo apt remove nginx`	Uninstall, keep config
`apt purge`	Remove package + config		`sudo apt purge nginx`	Full uninstall
`apt search`	Search repositories		`apt search editor`	Find packages
`apt list --installed`	List installed packages		`apt list --installed`	Inventory
`apt-get`	Lower-level APT frontend		`sudo apt-get install pkg`	Scripting-friendly APT
`dpkg`	Low-level .deb tool	`-i -l -r`	`dpkg -l | grep nginx`	Direct package operations
`snap`	Universal package manager	`install list remove`	`sudo snap install code`	Sandboxed packages
`flatpak`	Universal package manager	`install list`	`flatpak install app`	Sandboxed desktop apps

0.33.6 Processes & System Management

Command	Meaning	Key Options	Example	Purpose
`ps`	Snapshot of processes	`aux`	`ps aux`	List running processes
`top`	Live process monitor		`top`	Real-time resource view
`htop`	Enhanced live monitor		`htop`	Friendlier real-time view
`pgrep`	Find PID by name		`pgrep firefox`	Locate process ID
`pidof`	Find PID of a program		`pidof sshd`	Locate process ID
`kill`	Send a signal	`-9`	`kill -9 1234`	Terminate a process
`killall`	Kill by process name		`killall firefox`	Terminate all matches
`pkill`	Kill by pattern		`pkill -f script.py`	Pattern-based termination

Command	Meaning	Key Options	Example	Purpose
`jobs`	List shell jobs		`jobs`	Background/foreground jobs
`bg`	Resume job in background		`bg %1`	Continue paused job
`fg`	Bring job to foreground		`fg %1`	Resume interactively
`nohup`	Immune to hangup signal		`nohup ./script.sh &`	Survive terminal close
`nice`	Launch with priority	`-n`	`nice -n 10 cmd`	Lower scheduling priority
`renice`	Change running priority		`renice 10 -p 1234`	Adjust priority live
`systemctl`	Control systemd services	`start stop status enable`	`systemctl status nginx`	Manage services
`journalctl`	Query systemd logs	`-u -f -b`	`journalctl -u nginx`	View service logs

0.33.7 Storage

Command	Meaning	Key Options	Example	Purpose
`lsblk`	List block devices		`lsblk`	Show disks/partitions
`blkid`	Show filesystem UUIDs/types		`blkid`	Identify partitions
`mount`	Attach a filesystem		`sudo mount /dev/sdb1 /mnt`	Make storage accessible
`umount`	Detach a filesystem		`sudo umount /mnt`	Safely disconnect storage
`fdisk`	Partition tool (MBR/GPT)	`-l`	`sudo fdisk -l`	List/modify partitions
`parted`	Advanced partition tool		`sudo parted /dev/sdb`	Resize/create partitions

0.33.8 Networking

Command	Meaning	Key Options	Example	Purpose
`ip addr`	Show IP addresses		`ip addr show`	Interface/IP info
`ip link`	Show/manage interfaces		`ip link show`	Interface state
`ip route`	Show routing table		`ip route show`	Routing/gateway info
`ip neigh`	Show ARP/neighbor table		`ip neigh show`	MAC-to-IP mappings
`ping`	Test reachability	`-c`	`ping -c4 8.8.8.8`	ICMP echo test
`ss`	Show sockets	`-tulnp`	`ss -tulnp`	Listening ports/connections
`traceroute`	Show route hops		`traceroute example.com`	Path to destination
`tracepath`	Route hops, no root needed		`tracepath example.com`	Unprivileged traceroute
`dig`	DNS lookup (detailed)	`+trace`	`dig example.com`	Query DNS records
`nslookup`	DNS lookup (simple)		`nslookup example.com`	Basic DNS query
`host`	DNS lookup (concise)		`host example.com`	Quick DNS answer
`curl`	Transfer data via URL	`-v -I -o`	`curl -v https://x.com`	HTTP(S)/FTP requests
`wget`	Download files	`-O`	`wget https://x.com/file`	File retrieval
`arp`	Show/manage ARP table		`arp -a`	Legacy neighbor table view
`hostname`	Show/set hostname		`hostname`	Machine name
`nmcli`	NetworkManager CLI		`nmcli device status`	Manage network connections

0.33.9 SSH & Remote Access

Command	Meaning	Key Options	Example	Purpose
`ssh`	Secure remote shell	`-i`	`ssh user@host`	Remote login
`scp`	Secure copy	`-r`	`scp file user@host:/path`	Copy over SSH

Command	Meaning	Key Options	Example	Purpose
`sftp`	Secure FTP		`sftp user@host`	Interactive file transfer
`ssh-keygen`	Generate SSH key pair	`-t ed25519`	`ssh-keygen -t ed25519`	Create key-based auth
`ssh-copy-id`	Install public key remotely		`ssh-copy-id user@host`	Enable passwordless login

0.33.10 Bash & Text Processing

Command	Meaning	Key Options	Example	Purpose
`echo`	Print text	`-e`	`echo "hi"`	Output a string
`printf`	Formatted print		`printf "%s\n" "hi"`	Precise formatted output
`env`	Show environment variables		`env`	List env vars
`export`	Set environment variable		`export VAR=val`	Make var available to children
`grep`	Search text	`-i -r -v -E`	`grep -i error log`	Pattern matching
`cut`	Extract columns/fields	`-d -f`	`cut -d, -f2 file`	Slice delimited text
`sort`	Sort lines	`-n -r`	`sort -n file`	Order text
`uniq`	Remove/count duplicates	`-c`	`sort file | uniq -c`	Deduplicate
`wc`	Count lines/words/bytes	`-l -w -c`	`wc -l file`	Count text units
`tr`	Translate characters		`tr 'a-z' 'A-Z'`	Character substitution
`xargs`	Build commands from input		`find . -name "*.tmp" | xargs rm`	Chain input to a command
`tee`	Split output to file + stdout	`-a`	`cmd | tee out.txt`	Log while viewing
`sed`	Stream editor	`-i -e`	`sed 's/a/b/g' file`	Find/replace
`awk`	Pattern-scanning language		`awk '{print $1}' file`	Column processing

0.33.11 Bash Scripting

Command / Syntax	Meaning	Example	Purpose
`#!/bin/bash`	Shebang	first line of script	Specify interpreter
`read`	Read user input	`read name`	Capture input
`$1, $2, $@, $#`	Positional arguments	`echo $1`	Access script arguments
`$?`	Last exit code	`echo $?`	Check success/failure
`if/elif/else`	Conditional	`if [ -f file ]; then`	Branch logic
`for`	Loop over items	`for i in *.txt; do`	Iterate
`while`	Loop while condition true	`while read line; do`	Repeat until false
`function` / `name()`	Define reusable block	`myfunc() { ...; }`	Code reuse
`case`	Multi-branch match	`case $1 in ... esac`	Pattern-based branching

0.33.12 Logs & Troubleshooting

Command	Meaning	Key Options	Example	Purpose
`journalctl`	systemd log query	`-u -b -p -f`	`journalctl -p err -b`	Structured log review
`dmesg`	Kernel ring buffer	`-T`	`dmesg | tail`	Boot/hardware messages
`tail -f`	Live-follow a file		`tail -f /var/log/syslog`	Watch logs in real time

0.33.13 Security & Hardening

Command	Meaning	Key Options	Example	Purpose
`ufw`	Uncomplicated firewall	`allow deny enable status`	`sudo ufw allow 22`	Simple firewall control
`iptables`	Netfilter firewall (legacy)		`sudo iptables -L`	Direct packet filter rules
`nftables` (`nft`)	Modern netfilter frontend		`sudo nft list ruleset`	Current firewall standard
`fail2ban-client`	Brute-force protection control	`status`	`fail2ban-client status sshd`	View/manage banned IPs

Command	Meaning	Key Options	Example	Purpose
`aa-status`	AppArmor profile status		`sudo aa-status`	View confinement state
`aa-enforce`	Set AppArmor profile to enforce		`sudo aa-enforce /path`	Enable blocking mode
`aa-complain`	Set AppArmor profile to complain		`sudo aa-complain /path`	Log-only mode
`sha256sum`	SHA-256 checksum	`-c`	`sha256sum file`	Integrity verification
`sha512sum`	SHA-512 checksum	`-c`	`sha512sum file`	Stronger integrity check
`md5sum`	MD5 checksum (legacy)		`md5sum file`	Non-security dedup/compare only

0.33.14 Ethical-Hacking & Analysis Commands (Authorized Lab Use)

Command	Meaning	Key Options	Example	Purpose
`nmap`	Network/port scanner	`-sV -O -p`	`nmap -sV target`	Enumerate services (authorized targets only)
`tcpdump`	Packet capture (CLI)	`-i -w -c`	`tcpdump -i any -c 10`	Capture live traffic
`tshark`	Packet capture (Wireshark CLI)	`-i`	`tshark -i any`	Terminal packet analysis
`strings`	Extract readable text	`-n`	`strings binary`	Static binary inspection
`xxd`	Hex dump		`xxd file | head`	Byte-level file inspection
`hexdump`	Hex dump (alternate)	`-C`	`hexdump -C file`	Byte-level file inspection
`whois`	Domain registration lookup		`whois example.com`	Ownership/registration info

0.33.15 System Internals & Performance

Command	Meaning	Key Options	Example	Purpose
`strace`	Trace system calls	`-c -p`	`strace ls`	See syscalls a program makes
`lsof`	List open files/sockets	`-i`	`lsof -i :80`	See what's using a file/port
`vmstat`	Virtual memory stats		`vmstat 1 5`	CPU/memory/IO snapshot over time
`iostat`	Disk I/O statistics		`iostat -x 1`	Disk performance monitoring
`free`	Memory usage	`-h`	`free -h`	RAM/swap usage
`uptime`	System uptime + load average		`uptime`	Quick health check
`sar`	Historical system activity		`sar -u 1 3`	Trend-based performance data
`lsmod`	List loaded kernel modules		`lsmod`	Kernel module inventory
`modprobe`	Load/unload kernel module	`-r`	`sudo modprobe -r module`	Manage kernel modules
`sysctl`	View/set kernel parameters	`-w`	`sysctl -w net.ipv4.ip_forward=1`	Runtime kernel tuning
`ldd`	Show shared library dependencies		`ldd /bin/ls`	Inspect linked libraries

0.33.16 Miscellaneous Everyday Commands

Command	Meaning	Key Options	Example	Purpose
`history`	Show command history		`history`	Review past commands
`clear`	Clear terminal screen		`clear`	Tidy display
`date`	Show/set date and time	`+FORMAT`	`date +%A`	Current date/time
`cal`	Show a calendar		`cal`	Quick calendar view
`uname`	Kernel/system info	`-a`	`uname -a`	Identify kernel/architecture
`hostname`	Show/set hostname		`hostname`	Machine identity

Command	Meaning	Key Options	Example	Purpose
`reboot`	Restart the system		`sudo reboot`	Restart
`shutdown`	Power off / schedule shutdown	`-r -h now`	`sudo shutdown -h now`	Power off / restart with delay option
`crontab`	Manage scheduled jobs	`-e -l`	`crontab -e`	Edit recurring tasks
`man`	Manual pages		`man ls`	Full documentation for a command

Part VI

Appendix

0.34 GLOSSARY

AppArmor — A Linux kernel security module that confines individual programs to a predefined set of resources, using per-application profiles.

APT — Advanced Package Tool; the package management system used by Debian-family distributions (Ubuntu, Mint, Kali) to install, update, and remove software.

Bash — Bourne Again SHell; the default command interpreter on most Linux distributions.

cgroups — Control groups; a kernel feature that limits and accounts for resource usage (CPU, memory, I/O) of a process or group of processes.

CIDR — Classless Inter-Domain Routing notation (e.g., `192.168.1.0/24`) used to represent an IP network and its subnet mask compactly.

CLI — Command-Line Interface; interacting with software through typed text commands.

Daemon — A background process, typically started at boot and running without a controlling terminal, that provides an ongoing service (e.g., `sshd`, `cron`).

Distribution (distro) — A complete, installable operating system built around the Linux kernel, bundled with a package manager, utilities, and often a desktop environment.

DNS — Domain Name System; translates human-readable domain names into IP addresses.

FHS — Filesystem Hierarchy Standard; the specification defining the standard directory layout (`/etc`, `/var`, `/usr`, etc.) used by Linux distributions.

GID — Group Identifier; a numeric ID representing a group in `/etc/group`.

GNU — A project started by Richard Stallman to build a complete free Unix-like operating system; supplies much of the userland software (Bash, coreutils, GCC) used alongside the Linux kernel.

GPT/MBR — GUID Partition Table and Master Boot Record; two different partition table formats for disks, with GPT being the modern standard supporting larger disks and more partitions.

Inode — A data structure storing metadata about a file (permissions, owner, size, pointers to data blocks) — the “hard link” target.

IPv4/IPv6 — The two major versions of the Internet Protocol addressing scheme; IPv4 uses 32-bit addresses, IPv6 uses 128-bit addresses.

Kernel — The core software that manages a computer’s hardware, memory, and processes, and mediates access via system calls.

Kernel module — A piece of code that can be loaded into or removed from the running kernel without a reboot, typically a device driver.

LTS — Long-Term Support; an Ubuntu release supported with updates for an extended period (typically five years), recommended for production and beginner use.

Namespace (Linux) — A kernel feature that isolates what a process can see (its own view of processes, network interfaces, mounts, etc.), foundational to containers.

Package — A bundled unit of software (binaries, configuration, metadata) installable via a package manager.

PID — Process Identifier; a unique number the kernel assigns to each running process.

Repository — A server (or set of servers) hosting packages that a package manager can install from.

Root — The Unix/Linux superuser account (UID 0) with unrestricted system access; also refers to the top-level directory `/`.

Shell — A program that interprets and executes commands typed by the user (e.g., Bash, Zsh).

SSH — Secure Shell; an encrypted protocol for remote login and command execution.

Sticky bit — A permission flag on a directory that restricts file deletion to the file's owner, even if the directory is world-writable (used on `/tmp`).

Sudo — A command that allows a permitted user to execute another command with the privileges of another user (typically root), with logging.

SUID/SGID — Set User/Group ID; permission bits causing an executable to run with the file owner's/group's privileges instead of the invoking user's.

Systemd — The init system and service manager used by most modern Linux distributions, running as PID 1.

TCP/UDP — Transmission Control Protocol (reliable, connection-oriented) and User Datagram Protocol (fast, connectionless); the two main transport-layer protocols used over IP.

UID — User Identifier; a numeric ID representing a user account.

Unix philosophy — A design approach favoring small, single-purpose tools composed together via text streams and pipes, rather than large monolithic applications.

0.35 FURTHER RESOURCES

0.35.1 Official Documentation

- Ubuntu documentation and release notes (ubuntu.com)
- Linux Mint documentation (linuxmint.com)
- Kali Linux official documentation and tool listings (kali.org)
- The Linux kernel documentation (kernel.org)
- `man` pages — every command in this book has a full manual page (`man <command>`) with exhaustive detail beyond this book's scope.

0.35.2 Practicing Cybersecurity Skills Legally

- Intentionally vulnerable virtual machines designed for offline, isolated practice (search reputable security training VM repositories).
- Capture-the-Flag (CTF) platforms designed for legal skill-building, ranging from beginner-friendly to expert.
- Deliberately vulnerable web applications built for local practice of concepts in Part 22.
- Vendor-run bug bounty platforms, for those ready to test real systems under an explicit, scoped authorization agreement.

> ▲ **WARNING**
> Always verify a training platform's terms of use, and only test systems you own or are explicitly authorized to test. This book does not endorse or link to any specific unauthorized-access techniques or targets.

0.35.3 Where to Go Next

- **Systems administration track:** deepen Part 24's systemd coverage, learn configuration management tools, and study high-availability and backup architecture.
- **Cybersecurity track:** build on Parts 18–23 with structured coursework in network security, web application security, and a recognized certification path appropriate to your target role (SOC analyst, penetration tester, etc.).
- **Development track:** use your new Linux fluency to work comfortably in containerized environments (Docker/Kubernetes), CI/CD pipelines, and cloud infrastructure — nearly all of which assume the skills built in this book.

0.35.4 About This Handbook

This handbook is designed to be used twice: once read start to finish to build a complete mental model of Linux, and then indefinitely afterward as a reference — the cheat sheets (Part 29), master command reference (Part 33), and troubleshooting decision points (Part 16) are built for repeated lookup, not one-time reading. Keep it alongside your terminal.